I.U.T. Aix
Département génie informatique

Génie logiciel

1er trimestre 2002/2003 (semaines 7 à 11)

Cours Ada (suite)

troisième partie (cours 10 à 14 et numériques 1 et 2)

Daniel Feneuille

Nom étudiant :

Cours 10 Ada T.A.D, C.O.O, objets (4 heures)

Thème : Le T.A.D. la C.O.O. les objets

Avertissement (ce cours n°10 est renommé **hard** ! Et il l'est !) **:**

Après avoir étudié les **paquetages** et les thèmes, très importants aussi, que sont la **généricité** et les **exceptions** il est souhaitable de **préciser** les notions déjà bien introduites dans vos esprits tels que : le type abstrait de données (**T.A.D.**) d'une part, et la conception orientée objet (**C.O.O.**) d'autre part (avec modules et objets).

Le lecteur averti de ces concepts pourra trouver la présentation qui va suivre un peu « simpliste » ou réductrice. Il faut savoir que cet enseignement se situe dans la **septième semaine** du cursus du DUT qui en contient 56 ! Relativisez ! Mais, même si c'est un peu tôt, il est temps cependant de mieux préciser les notions citées plus haut qui sont des **fondements du génie logiciel**. C'est une étape, **hélas**, que les étudiants ne goûtent pas toujours car elle **nécessite un gros effort d'abstraction** éloigné du bidouillage (voire du maquettage) qu'ils affectionnent tant.

Nous sommes persuadés, aussi, que l'on peut très bien tenir un discours (indispensable) à forte imprégnation de génie logiciel et de démarche qualité du logiciel, avec des techniciens supérieurs, sans être obligé d'utiliser des démarches trop théoriques. Enfin le langage Ada, même s'il nous plait tant, n'est pas incontournable [1]. La façon d'enseigner est sûrement aussi importante que le langage lui même. Par exemple méditons ceci :

- Je prête l'oreille et puis j'oublie.
- J'écoute et je m'intéresse.
- Je vois et je me souviens mieux.
- Je fais et je comprends tout.

← Extrait d'un maxime chinoise dont nous nous inspirons pour enseigner Ada.

Retour sur le typage et surtout le fort typage (en guise d'introduction) :

Au début était le type ! [2] En effet, dès l'apparition des premiers langages de programmation (le `FORTRAN` par exemple) les programmeurs ont disposé de la notion de **type** (prédéfinis seulement !). Les plus usités étaient les réels, les entiers et les booléens. Le type des littéraux **était déduit** de leur « structure » (ou dénotation) ! Ainsi a priori `3.14159` « était » un réel (évident ! Quoique ! Voir plus bas !). Mais les déclarations de variables n'étaient pas toujours obligatoires ! C'est grave docteur ! Par exemple en `FORTRAN` toutes les variables, non déclarées, mais commençant par les lettres de l'intervalle `I..N` (I, N comme … `Integer` !) étaient des entiers, les autres (non déclarées) étaient des réels. Soit ! Mais il n'y avait pas de vérification syntaxique pouvant pallier des erreurs conceptuelles (comme, heureusement, Ada sait si bien le faire). Par exemple en `FORTRAN` :

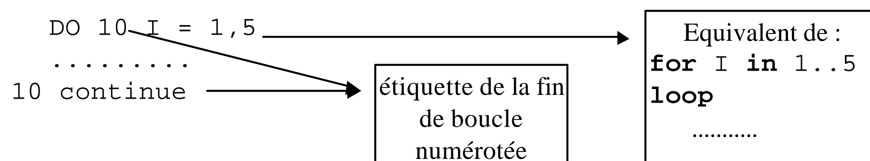
```
I = 3.14159
```

est une instruction d'affectation [3] qui peut « déclarer » implicitement et initialiser ainsi la variable **entière** I (en lui affectant la valeur 3). Affreux, affreux ! Mais nous sommes dans les années 50 ! Indulgence !

Une anecdote véridique maintenant. Dans les années 60 la sonde de la NASA destinée à observer la planète Vénus **passa très loin** de son objectif à cause d'un « erreur » due à la permissivité du typage. Voyons cela.

[1] Quoique je me sens incapable d'enseigner le génie logiciel avec du C !
[2] Je paraphrase là une célèbre citation que tout le monde aura, je l'espère, reconnue !
[3] oui, **affectation** car en FORTRAN (comme en C ou en C++ ou en Java) **c'est le signe =** qui symbolise cette instruction alors qu'en Ada (comme en Pascal et en Algol) c'est `:=`.

Pour faire une boucle en `FORTRAN` on écrit :



Les programmeurs avaient écrit :

`DO 10 I = 1.5` au lieu de `DO 10 I = 1,5`

Notez bien la **présence du point au lieu de la virgule**. Et bien le compilateur a traduit ceci :

`DO10I = 1.5` soit l'affectation de `1.5` à la variable `DO10I` [4]

C'est le `1.5` qui a conditionné l'affectation (d'autant que, en FORTRAN, les espaces entre les « mots » ne sont pas toujours significatifs). Même en phase de « **revue de code** » personne n'a vu l'erreur, tout le monde a cru à une boucle `DO`. La séquence (marquée `.........`) ne fut jamais exécutée 5 fois mais .... une seule fois!

Cette erreur historique, on l'aura compris, nous permet de noter qu'il faut absolument, non seulement une syntaxe rigoureuse (que les compilateurs doivent respecter et non interpréter), mais aussi un **fort** typage (nous y reviendrons dans les cours numériques 1 et 2). Mais cela ne suffit pas. On a bien vu (le 4 juin 1996) le premier tir de la fusée Ariane 5 faire un « flop » retentissant (bien que programmé en Ada !). Heureusement le langage Ada n'était pas foncièrement en cause ! C'est l'utilisation des exceptions qui a été occultée ! Ouf !

En **résumé**, il est indispensable, en programmation saine, de **tout déclarer** sans ambiguïté. Non seulement les variables, mais **aussi les types** que l'on doit **pouvoir construire** en les **nommant**, en les **décrivant** et en leur **associant** des « opérateurs » ou méthodes (suivant en cela le principe de **l'approche objet** vue plus loin). C'est le **typage** des données. Le typage permet de documenter le rôle et la signification des objets utilisés. Le fort typage (même s'il agace certains développeurs) permet de confier au compilateur des vérifications (d'essence sémantique !). La possibilité de créer ses propres types permet d'augmenter le niveau de sémantique de l'application. En pratiquant ainsi on assure une **bonne sécurité** de développement des logiciels. En Ada, on l'a déjà bien vu en TD et TP (par exemple le n° 10), l'outil idéal pour remplir ce premier principe c'est le **paquetage** structurant le T.A.D.

Le type abstrait de données (T.A.D.) « cas particulier de module » :

Un **type abstrait de données** (quel que soit le langage) c'est le **rassemblement de toutes les entités** ayant un lien « logique » entre elles. Ce type est clairement identifié par un nom (**identificateur**). Implicitement il lui est attaché des propriétés sous-jacentes et même explicitement (**spécifications**) souvent bien décrites. Par exemple l'affectation sera permise (ou non permise : cas d'un type **`limited`**), la structure du type sera accessible (ou non accessible : cas d'un type **`private`**), etc. L'exemple le plus académique et qui est développé dans tous les bons ouvrages (en Ada ou non) est le type `PILE`. Nous y avons passé 4 heures (TD-TP n°10 la semaine dernière) en suivant bien sa construction de plus en plus complexe (par itérations critiques !) pour arriver à un type très acceptable et utile pour la suite. Un autre exemple est le type `T_Rationnel` (TD-TP n°11 cette semaine). Sans oublier les `Bounded_String` (étudiés au TD-TP n°12 dès la semaine prochaine).

Notez bien les principes fondamentaux qui vont suivre et leur mise en œuvre pendant tous ces TDTP.

[4] non déclarée mais commençant par un `D` donc ..... un réel recevant `1.5` comme littéral!

Voyons trois **principes méthodologiques** pour concevoir un « **bon** » type abstrait de données (T.A.D.).

Premier principe : **<u>séparer</u> la description du T.A.D. de sa réalisation.**

L'idée est de **ne montrer**, aux utilisateurs, **que ce qui est nécessaire**; c'est-à-dire les propriétés **externes**. Le détail des **réalisations** (ou **implémentations**) est sans intérêt. Par exemple (voir cours n° 6, 7 et 8 pour le type `T_COMPLEXE`) il importe peu à l'utilisateur de savoir comment s'effectue la division de deux variables complexes (les formules sont connues); l'essentiel est que soit bien décrit : l'opérateur / et la façon de l'utiliser. Cette « séparation » est bien sûr possible avec le découpage du paquetage en deux parties et si possible compilées séparément. La partie **réalisation** n'étant pas « donnée » à l'utilisateur (pas de sources). Mieux encore, puisque l'on cache la réalisation des spécifications, il est souhaitable de « cacher » *<u>aussi</u>* la **réalisation de la structure de données** elle même. On l'aura compris c'est en Ada grâce à la déclaration **`private`** (voire **`limited private`**) que l'on réalisera cette propriété de « masquage » de la donnée.

Deuxième principe : **<u>encapsuler</u>.**

Il importe de rendre « **solidaires** » la **structure de données et ses méthodes** (sous-programmes). C'est en déclarant **ensembles** (toujours dans les spécifications du paquetage) ces entités que l'on « **attache** » le type et ses « opérateurs ». Le but est aussi **d'interdire**, à l'utilisateur, **l'accès aux composants** de la structure de façon à garder **l'intégrité** des données (contrôle complet de la manipulation des objets et de leurs valeurs). A charge alors de fournir, à l'utilisateur, des « outils » supplémentaires pour initialiser, modifier, observer, parcourir, etc. les instances (ou réalisations concrètes) du type en question. On parle alors respectivement de **constructeurs**, **modifieurs**, **accesseurs (ou sélecteurs)**. On peut aussi réaliser d'autres « outils » comme **conteneurs**, **itérateurs**[5], **destructeurs** etc. C'est avec la séparation en deux parties (et compilation séparée) et la privatisation que l'on réalise l'objectif **d'encapsulation** [6].

Troisième principe : **« <u>adapter</u> » le type abstrait.**

Adapter le type abstrait c'est :

- Permettre des « copies » **identiques mais incompatibles entre elles**. On (re)**verra** cela avec le principe de la **dérivation** avec **`new`** et l'héritage direct.
- Permettre des **instanciations** « différentes » du type abstrait grâce aux « paramètres ». On **a vu** cela avec la **généricité** qui permet de définir des modèles (des moules) facilitant la réutilisabilité.
- Permettre la signalisation et le traitement des **erreurs**. On **a vu** cela avec les **exceptions**.
- Permettre **l'ajout de fonctionnalités** sans modifier l'existant. On **a vu** cela avec les paquetages hiérarchiques.
- Permettre une **évolution** de la structure de données. On **verra** cela avec la dérivation de « **l'objet** » et les **classes** (fin de ce cours n°10).

Bref il y a du pain sur la planche. Ces grands principes sont aussi partie prenante dans la notion de **conception orientée objet** (C.O.O.) que nous allons voir.

La conception orientée objet (C.O.O.).

Le terme « orienté objet » a fait, par le passé, couler beaucoup d'encre et continue à opposer des « chapelles » d'informaticiens. En simplifiant : sous cette appellation on désigne **deux idées** assez différentes.

1. On a d'une part, la conception (assez récente) où prédomine « l'objet vrai » en tant qu'unité de structuration principale des algorithmes. Ceci implique le concept sous-jacent de **classes** susceptibles d'être **héritées**. Les langages permettant les objets sont : SmallTalk, C++, Eiffel, Java et Ada95 (liste non exhaustive !). On parle souvent à cet égard de **programmation objet** (avec mécanisme d'héritage). On étudie cet aspect avec Ada au premier trimestre et à partir du deuxième trimestre avec le langage C++ (et même en deuxième année).

---

[5] Voir notamment cette notion dans le livre de Rosen (Méthodes de génie logiciel avec Ada95) page 120.

[6] Un autre intérêt de la séparation (spécifications et corps) est de permettre une organisation optimale du développement des logiciels. En effet la réalisation (corps) peut être programmée par une équipe pendant qu'une autre équipe continue de développer en s'appuyant sur les spécifications supposées réalisées.

2. D'autre part, la conception où prédomine le concept de type abstrait de données (vu précédemment) **sans qu'interviennent** les notions « vraies » d'héritage et de classes. C'est G.BOOCH (en 1988) qui fut l'instigateur de cette approche. Ada83 et ses paquetages étaient déjà bien adaptés à cette approche nous allons nous y attarder quelques instants avant de voir les objets et les classes (voir plus loin).

Jean-Pierre ROSEN [7] a proposé aussi une distinction entre ces deux idées en se plaçant cette **fois du point de vue de la méthode de conception d'un programme** :

- par **classification** pour la première idée « objet » (vue plus loin, bis !),
- par **composition** pour la deuxième idée c'est le « module ».

Mais, tout d'abord : pourquoi une méthode de conception orientée objet (quelle qu'elle soit) ?

Deux objectifs au moins, guidés par le même fil directeur : **diminuer les coûts** :

- de **production** d'une part,
- de **maintenance** d'autre part.

Il peut paraître paradoxal d'avoir mis au même niveau de préoccupation ces deux coûts. En première estimation on pourrait imaginer que c'est la production de logiciel qui coûte le plus ! le code à générer étant volumineux. `IL N'EN EST RIEN!` L'expérience montre que **la maintenance coûte plus** (ou au moins aussi) **cher**.

Dans le concept de **maintenance** on distingue :

- la maintenance **corrective** d'une part,
- la maintenance **adaptative** d'autre part.

La maintenance **corrective** c'est l'étape (jamais finie !) qui consiste à **réparer** les erreurs qui restent toujours dans le « code ». **Erreurs de compilation** évidemment, mais aussi **erreurs conceptuelles** quand les tests ne donnent pas satisfaction. Quand une erreur est **vite localisée** sa réparation ne coûte pas grand chose. Quand il faut des heures **ou des jours** pour y parvenir c'est autre chose. L'erreur trouvée est souvent **la même un peu partout** car on a souvent, **maladroitement**, dupliqué le code en question ! La bonne technique consiste à **factoriser** ce code (généricité si possible) pour mieux le **dériver** (ou instancier) ensuite. Un seul endroit à réparer dans ce cas ! Economie en temps et en fiabilité (re-compilation oblige quand même !).

La maintenance **adaptative** consiste à modifier (légèrement ou beaucoup) le logiciel produit car rien n'est immuable. Si le logiciel est « **flexible** » tout va bien, si c'est une « usine à gaz» bonjour les dégâts ! L'outil de décomposition en éléments que nous appellerons **modules** (regroupement d'entités logiques [8]) ou analyse modulaire permet (avec le **faible couplage** des modules) une excellente flexibilité.

Quant à la **production à moindre coût** elle est favorisée par la **réutilisation** de composants logiciels (achetés ou déjà mis au point) bien **testés** donc considérés comme validés.

La réutilisation.

Pour construire des applications informatiques (fiables et conséquentes) de nombreuses entreprises **utilisent** et **réutilisent** des composants logiciels. On peut voir cette activité comme un simple **« assemblage » de composants** mais cette technique va bien plus loin et nécessite de la **méthode**, des **outils** et même induit de nouveaux métiers (créateurs, bibliothécaires, experts ou architectes). Mais pour être efficace la réutilisation ne doit pas être occasionnelle ! La réutilisation doit structurer le projet et elle met à mal les attitudes des programmeurs. Enfin, encore un obstacle, elle n'est, hélas, pas rentable tout de suite !

---

[7] le lecteur, avide d'en savoir plus, lira son livre (en bibliothèque) « Méthodes de génie logiciel avec Ada95 ». Mais il me semble déjà avoir fait les louanges de ce livre !
[8] vous avez dit paquetage par exemple?

L'objectif premier est bien sûr **l'amélioration de la productivité** du logiciel (respect des délais, optimisation de la qualité, diminution des coûts) mais ce n'est pas facile. Il faut souvent « **imposer** » cette culture dans l'entreprise qui s'oppose au syndrome du NIH (not invented here). Voir en page 9 des compléments.

Le cheminement traditionnel (concernant ces composants) est schématiquement celui-ci :

Identifier ⇒ Fabriquer ⇒ Certifier ⇒ Archiver ⇒ Retrouver ⇒ Adapter ⇒ Maintenir.

Le module.

Dans la logique développée ci dessus un module est donc un **modèle** (informatique) associé à un élément du monde réel (à informatiser). Par exemple : un écran, une feuille de paie, un moteur, un polynôme, une pile, etc. C'est un **modèle abstrait** (une vue abstraite) qui **dépasse** l'aspect uniquement fonctionnel. C'est **l'unité de structuration conceptuelle**. Un module est le rassemblement, en une seule entité, de toutes les manipulations de cette abstraction caractérisée par son type [9]. Si l'on sépare les responsabilités de l'utilisateur du module de celles du concepteur on favorise la flexibilité tant prisée plus haut [10].

Décrire une abstraction (sous forme d'un « module »).

Schématiquement on distingue dans une abstraction du monde réel :

- L'information proprement dite,
- les services que l'on peut en attendre,
- le comportement de « l'objet » (objet est pris ici comme synonyme du type).

Ceci peut se traduire (voir plus haut) par un type abstrait de données [11] (T.A.D.) avec respectivement :

- la déclaration du type et de la structure de données associée (synonyme : **donnée-membre**).
- les sous-programmes (procédures et fonctions) **ayant au moins le type comme paramètre**. Ces services sont en gros soit des **renseignements** sur le type, soit des **transformations** sur celui-ci. (synonyme : **méthodes**).
- Des commentaires pour décrire la normalité du comportement, des exceptions (cours n°8) pour gérer les « accidents », voire des tâches [12] pour rendre l'objet sous-jacent « indépendant » (objet actif).

Relations entre les modules.

On distingue deux « démarches » qui privilégient :

- soit l'analyse par **composition** (ou par décomposition suivant la manière !),
- soit l'analyse par **classification** (appelée aussi analyse par analogie ou analyse différentielle).

L'analyse par (dé)**composition** c'est la construction d'une architecture de programme par approche (descendante ou ascendante). On a déjà vu cette notion (de façon simplifiée) avec la **programmation structurée** (cours algorithmique) : l'ancêtre des méthodes dans les années 70. Mais dont on a dit les limites avec les tailles gigantesques des logiciels actuels. On « assemble » les modules soit par raffinement (du plus général vers le plus élémentaire) c'est la décomposition, soit par assemblage « vrai » du plus simple vers le plus compliqué (c'est la composition de ressources). Et souvent on ne peut s'empêcher de mêler les deux manières.

L'analyse par **classification** c'est une démarche qui consiste à voir une abstraction comme « **voisine** » d'une autre déjà créée. L'abstraction nouvelle est alors :

[9] avez-vous encore dit paquetage?
[10] le paquetage vous dis-je!
[11] oui j'ai bien dit paquetage!
[12] vues au deuxième trimestre (ainsi que les objets protégés : objet passif).

- soit à compléter (c'est une spécialisation de la première),
- soit une généralisation de la première,
- soit une « collatérale » de la première (toutes les deux seront une spécialisation d'une troisième à créer).

Dans les deux démarches d'analyse (composition ou classification) la difficulté consiste toujours à « voir » les abstractions et leurs relations le plus tôt possible! **Mais ce n'est pas facile**. De l'expérience, de la discipline facilitent cette activité. On réfléchit beaucoup, on agit ensuite! C'est bien sûr ce que vous faites toutes et tous (gag !), mais ce n'est pas ce que fait votre voisin, votre collègue, voire votre chef (chef, oui chef !).

Quelques avantages de la méthode.

1) L'objet (ou module suivant les approches) **regroupant** les différentes caractéristiques de l'abstraction il sera beaucoup plus **facile de localiser** les endroits où il conviendra « d'intervenir » en cas de **maintenance**. Rien n'est éparpillé, tout est centralisé et lié.

2) La **réutilisation** est facilitée quand **on développe des logiciels connexes**. Les modules ou les objets peuvent être soit :

- réutilisés comme tels (ou avec très peu d'adaptation),
- modifiés par ajout de fonctionnalités et/ou de composants (en gardant invariants les autres éléments).

3) La **sécurité** de développement est garantie puisque le logiciel manie des objets informatiques abstraction des objets réels. Avec ce même état d'esprit la réutilisation (toujours elle) est facilitée [13].

4) Souvent (mais nous verrons cela plus tard au deuxième trimestre) les objets réels peuvent évoluer en **parallèle**. Si le langage permet cette prise en compte (tâches ou objets protégés en Ada, thread en C++ ou en Java) alors les objets informatiques associés évolueront de la même manière.

5) Le logiciel étant organisé en **couches logicielle hiérarchisées** il est possible d'établir le graphe des dépendances logiques entre les modules on parle de **topologie de programme**. La lisibilité du graphe (arbre ou éventuellement graphe acyclique) permet une vue synthétique de l'organisation des composants.

Remarques :

- Je redis ici combien pour le moment cette partie du cours est une « première couche » à la **difficile notion** de **conception objet**. Je n'ai pas parlé (ou presque pas) du concept **d'héritage** qui, pour simple qu'il puisse paraître, recouvre une somme importante de difficultés que nous verrons plus loin.

- Nous allons présenter sur les deux pages suivantes quelques notions complémentaires (notions certainement revues et complétées dans d'autre cours notamment en ACSI).

- A propos de cycle de vie celui présenté page suivante (dit en V ou AFCIQ) n'est pas unique mais c'est le plus connu et il est intéressant sur le plan méthodologique. Citons aussi dans la classe des cycles de vie dit progressifs le cycle en cascade. On découvre dans le cycle en V que le codage (et cela surprendra les étudiants) arrive **très tard** dans la démarche. Il est clair, aujourd'hui, que l'apparition de techniques à base d'abstraction et de généricité permet d'envisager le codage un peu plus tôt.

- En ce qui concerne les cycles de vie plus orientés vers le dialogue avec les futurs utilisateurs on peut citer les appellations : de cycle en spirale, de prototypage, de RAD. Ces techniques semblent susciter un certain engouement dans la profession. Mais elles engendrent souvent des catastrophes retentissantes. Attention donc ! Actuellement une ***notation*** intéressante pour analyser et spécifier à le vent en poupe c'est U.M.L. A voir !

---

[13] Encore que : revoir l'échec d'Ariane 5 où on réutilisa des composants d'Ariane 4 sans grandes précautions!

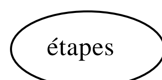
Le cycle de vie du logiciel.

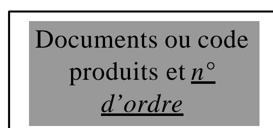
Il s'agit de présenter schématiquement les différentes étapes de la vie du logiciel ainsi que les relations possibles entre ces étapes. En général on distingue : l'analyse, la conception, l'implémentation, la livraison et la maintenance. Plus précisément et en détaillant on trouve :
Expression des besoins ⇒ spécifications du logiciel ⇒ conception générale ⇒ conception détaillée ⇒ codage ⇒ tests unitaires ⇒ tests d'intégration ⇒ validation ⇒ suivi d'exploitation.

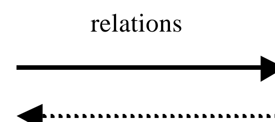
Ces 9 étapes sont souvent plus ou moins marquées et respectées. C'est dommage ! je ne parle même pas des équipes qui ne pensent qu'au codage ! De telles pratiques ne durent pas longtemps tellement le logiciel résultant est instable (usine à gaz!). Une présentation plus agréable que celle ci-dessus (trop linéaire) est la classique présentation du cycle en V (commentée en cours) qui laisse bien apparaître toutes les relations entre ces 9 étapes.

Expression des besoins
Suggestions d'amélioration *9*
Suivi d'exploitation
Cahier des charges *1*
Logiciel livré *8*
Spécifications du logiciel
Document de validation *2*
validation
Dossier d'analyse
Modules intégrés *7*
Conception générale
Dossier de tests
Tests d'intégration
Dossier de conception générale *3*
Modules testés unitairement *6*
Conception détaillée
Dossier de tests *4*
Tests unitaires
Dossier de conception détaillée *4*
Code des modules *5*
codage

Légende :







Compléments.

Statistiques inquiétantes.

Si vous n'êtes pas encore convaincus que le développement de logiciels nécessite de la méthode et des techniques fortes et structurées voici les résultats d'une étude récente (publiée par Standish Group en 2000) :

Sur un échantillon de 365 entreprises et organismes totalisant 8380 applications informatiques on constate que :

- 16,2% seulement des projets analysés sont **conformes** !
- 52,7% subissent des **dépassements de coût et de délais** d'un facteur de 2 à 3 accompagnés d'une diminution des fonctionnalités prévues !
- 31,1% sont tout simplement **abandonnées** en cours de développement !

Un peu de la méthode donc !

Documentation.

On verra dans le livre de Rosen un modèle (extrêmement riche) de **fiche de description** de composant logiciel (pages 394 à 396).

Réutilisation et composants logiciels [14].

L'idée qui sous-tend la notion de réutilisation est d'écrire le moins possible de code ! On y parvient soit en utilisant des générateurs de code source (voir par exemple glade avec gtkada pour construire des interfaces graphiques) soit en utilisant des composants logiciels.

Dans cette dernière idée (séduisante) de composants logiciels il s'agit de « copier » l'industrie des composants électroniques où l'ou assemble des briques (composants simples) pour obtenir un composant plus compliqué. Il ne viendrait à personne l'idée de re-fabriquer ou re-concevoir un transistor quand on monte une carte avec un micro-processeur. En programmation il est donc clair qu'il faut faire pareil : on ne ré-écrit pas un module qui existe et qui tourne ! Eh bien, cette idée toute simple ne « prend pas » et n'est pas appliquée (ou rarement : seulement sur des cas simples). Pourquoi ?

1. Le programmeur préfère refaire qu'utiliser même si le composant existe et est connu ! C'est passionnel !
2. Le programmeur préfère commencer à coder que réfléchir et spécifier ! C'est incurable !
3. Si le composant est possible il faut chercher l'oiseau rare ! C'est compliqué !
4. Si une liste de composants est disponible il faut choisir celui qui convient ! C'est déroutant !
5. Quand on croit tenir le composant on hésite encore ! Et s'il était bogué ? C'est évident !
6. Finalement, il coûte trop cher ce composant ! Imparable !

Les réflexions ci dessus frisent la caricature ? Un peu, mais pas vraiment ! On rencontre, plus souvent qu'on ne pense, ce genre de frein.

Ada est-il bien adapté pour écrire des composants réutilisables ? Evidemment l'Adatollah que je suis va répondre par l'affirmative mais en se justifiant.

- Ada est portable (car Ada est normalisé et les compilateurs respectent la norme !).
- Les paquetages Ada facilitent les spécifications (sans s'intéresser à la réalisation).
- Les génériques Ada offre une grande souplesse d'adaptation du composant.
- La surcharge facilite la lisibilité et la compréhension.
- Les clauses comme renames, with et use ajoutent à la facilité d'utilisation.

Alors pourquoi pas vous ?

[14] Ce petit chapitre est « inspiré » d'un article ancien (1990) de Bruno Caneill (L.L.A. n° 37 & 38)

Les objets enfin ! (l'héritage, la dérivation, l'extension de données, le polymorphisme, la liaison dynamique, les classes). Ouf !

Remarquons que certains concepts forts, intrinsèques à « l'objet » (à définir) étaient déjà présents dans Ada83 :

- **abstraction** grâce aux types privés et limités privés,
- **encapsulation** grâce à la séparation spécifications/corps et la compilation séparée,
- **généricité** qui permet de factoriser des parties de code pour des types différents,
- **héritage** (même s'il est assez pauvre!) grâce à l'instruction`new`,
- **classification** (même très pauvre) avec les sous-types (introduction d'une contrainte) et les types non contraints (permettant de créer une sur-classe).

Ada, dans sa révision, a dû tenir compte des fondements de Ada83 pour réaliser **l'héritage vrai**. Les paquetages (permettant encapsulation et modularité) **restent les bases** auxquelles on ajoute l'instruction de dérivation `new` mais **élargie aux types étiquetés**. Avec Ada95 on passe d'un langage à rudiments d'objets (Ada83) à un langage permettant les objets. Certains grincheux [15] trouveront encore à redire car ils ne retrouveront pas « leurs objets à eux » (les seuls qui vaillent bien sûr !). L'ennui naquit, un jour, de l'uniformité !

Qu'est-ce qu'un objet (humour !) ?

Un objet est une capsule logicielle oblative avec un tropisme conatif dont l'hétéronomie est la marque de la durée de l'éphémère et de la hoirie ![16]

Si cette citation (trouvée dans une pub Aonix) et attribuée à un universitaire niçois, est sûrement correcte (merci mon dictionnaire !) cette citation donc (compte tenu de notre culture littéraire de pseudo-scientifiques) ne nous avance pas à grand chose et mérite d'être explicitée ! Alors peut-être plus simplement en traduisant : un objet est un ensemble « logiciel » qui propose ses services à autrui, qui est capable de réagir à une demande, qui manque d'autonomie, qui a une durée de vie limitée et qui peut être hérité. Est-ce que cela va mieux ? Pas encore ! Pas étonnant ! On continue alors autrement. Un objet (sous entendu maintenant une ensemble logiciel qui est une notion claire !) se caractérise par des **demandes** qu'il sait satisfaire. Il dispose pour cela de **méthodes**. Si plusieurs objets savent satisfaire une même demande alors de ce point de vue ils se ressemblent, mais il n'est pas sûr qu'ils appliquent la même méthode pour y parvenir. Un ensemble de demandes permet de définir un modèle d'objets que l'on appelle alors une **classe**. Une classe est un ensemble d'objets (ou type pour simplifier) qui sont tous issus (ou dérivés) d'un même ancêtre (racine de la classe). En Ada cet ancêtre est obligatoirement un type étiqueté (`tagged`) même vide ! Cette contrainte permet de rester cohérent avec Ada83 tout en l'élargissant. En Ada **une classe se définit par un paquetage** déclarant un type (étiqueté) et des sous-programmes ayant ce type en paramètre. Cela ne va toujours pas mieux ? Bigre ! Voyons encore. Ada va permettre :

- **L'extension de données** grâce aux types étiquetés et à l'instruction `new`. On pourra aussi ajouter des méthodes (un peu différent de l'extension de fonctionnalités sur un même type « paquetages hiérarchiques »).
- **Le polymorphisme** qui est la capacité d'un objet à réagir, selon sa classe spécifique, à toute demande.
- **La liaison dynamique** permettant de différer le choix de la méthode jusqu'au moment de l'exécution (et pas à la compilation trop statique). Cette technique n'exige pas de pointeurs comme d'autres … (no comment !).
- **La finalisation** qui définira des sous-programmes spécifiques très utiles aux étapes clés de la vie d'un objet (création, affectation, destruction). Ce seront les types contrôlés (`controlled`).

Ada95 permet ces propriétés non pas en bloc (comme certains langages!) **mais à la demande** ! Intéressant ! Si vous n'avez pas encore tout compris rassurez vous car la notion d'objets **n'est pas si facile à comprendre**. Même J.G.P. Barnes le reconnaît ! Alors ! Cependant ce n'est pas parce que c'est difficile qu'il faut baisser les bras. En vous accrochant vous préparez votre avenir. Avenir à court terme puisqu'à partir du deuxième trimestre et en deuxième année vous allez beaucoup travailler avec C++ (qui est le plus répandu des langages à objets). Avenir à plus long terme puisque dans votre vie professionnelle si vous n'avez pas la culture « objets » vous stagnerez dans des développements d'applications dites « alimentaires » (non négligeables certes mais ...).

[15] Je n'ai nommé personne mais j'ai fortement pensé aux tenants d'un langage enfin normalisé !
[16] Comme disait ma grand mère : un intellectuel est quelqu'un de rassuré quand il n'est pas compris !

Un exemple (enfin!) pour montrer l'intérêt du concept d'objet.
Il est inspiré du livre « programmer en Ada 95 » de Barnes dont j'ai déjà fait la pub ! Nous verrons en TD-TP 13 un exercice plus intéressant et peut être plus original.
Soit un système de réservation de billet d'avion (à gérer). En Ada83 on pourrait avoir le TAD suivant :

```
package P_RESERVATION_83 is
   type T_CATEGORIE is (TOURISME, AFFAIRE, LUXE);
   type T_SITUATION is (COULOIR, FENETRE);
   type T_REPAS is (VEGETARIEN, VIANDE_BLANCHE, VIANDE_ROUGE);
   type T_VOITURE is (RENAULT, PEUGEOT, CITROEN);
   .......
   type T_RESERVATION (CAT : T_CATEGORIE) is
      record
         NUMERO_VOL : T_NUM_VOL;
         DATE_VOYAGE : T_DATE;
         NUM_SIEGE : T_NUM_SIEGE;
         SIEGE : T_SITUATION;
      case CAT is
           when TOURISME => null;
           when AFFAIRE | LUXE =>
              REPAS : T_REPAS;
              case CAT is
                 when TOURISME | AFFAIRE => null;
                 when LUXE => VOITURE : T_VOITURE;
              end case;
         end case;
      end record;
   ......
   procedure RESERVER (R : in out T_RESERVATION);
   procedure CHOIX_SIEGE (R : in out T_RESERVATION);
   procedure CHOIX_REPAS (R : in out T_RESERVATION);
   procedure CHOIX_VOITURE (R : in out T_RESERVATION);
   ........
end P_RESERVATION_83;
```

On peut remarquer qu'il n'est pas fait état ici de la notion de type privé on pourrait le faire mais l'intérêt est ailleurs (nous le ferons plus loin). On devine qu'il y a trois sortes de passagers qui ne bénéficient pas des mêmes prestations. La classe Touriste a droit à un siège (quand même !), les classes Affaire et Luxe ont droit à un repas et enfin seule la classe Luxe a droit à une voiture à l'arrivée. On constate un «codage » assez compliqué avec des structures de choix très imbriquées qui ne faciliteront pas la maintenance. En effet imaginons la création d'une quatrième catégorie `SUPERSONIQUE`. Il faut tout **modifier, rajouter des cas et tout recompiler**. Les sources d'erreurs ne sont pas négligeables même en Ada ! Et où est la réutilisabilité dans tout cela ?

Une solution Ada95 (remarquez bien le type article étiqueté « `tagged` »[17]).

```
package P_RESERVATION_95 is
   type T_SITUATION is (COULOIR, FENETRE);
-- le type T_CATEGORIE n'est plus nécessaire !
   type T_RESERVATION is tagged -- le discriminant a disparu!
      record
         NUMERO_VOL : T_NUM_VOL;  -- plus de case !
         DATE_VOYAGE : T_DATE;
         NUM_SIEGE : T_NUM_SIEGE;
         SIEGE : T_SITUATION;
      end record;
```

Notez le **tagged** en plus !

[17] Ce type avait été annoncé dans le cours n°6 sur les articles.

```
   procedure CHOIX_SIEGE (R : in out T_RESERVATION);
   procedure RESERVER (R : in out T_RESERVATION);
   type T_RESERVATION_TOURISME is new T_RESERVATION with null record;

   type T_REPAS is (VEGETARIEN, VIANDE_BLANCHE, VIANDE_ROUGE);

   type T_RESERVATION_AFFAIRE is new T_RESERVATION_TOURISME with
     record
          REPAS : T_REPAS;
     end record;

   procedure CHOIX_REPAS (R : in out T_RESERVATION_AFFAIRE);
   procedure RESERVER (R : in out T_RESERVATION_AFFAIRE); -- surcharge

   type T_VOITURE is (RENAULT, PEUGEOT, CITROEN);

   type T_RESERVATION_LUXE is new T_RESERVATION_AFFAIRE with
     record
          VOITURE : T_VOITURE;
     end record;

   procedure CHOIX_VOITURE (R : in out T_RESERVATION_LUXE);
   procedure RESERVER (R : in out T_RESERVATION_LUXE); -- surcharge

end P_RESERVATION_95;
```

Remarques :

- On a la possibilité de dériver en cascade dans les spécifications et bien sûr d'hériter immédiatement.
- Avec (**with null record**) la «réservation tourisme » est identique à la réservation de base, ce qui signifie dans le problème réel que la catégorie tourisme n'a pas de prestation (ni repas ni voiture).
- Le type étiqueté (**tagged**) est **extensible par dérivation** (c'est la **racine de la classe**). Notez aussi le **with** associé à **record** (qui n'a rien à voir avec le **with** évoquant une ressource paquetage). Les types dérivés à nouveau **ne comportent plus** la mention « étiquetée » **tagged**.
- La dérivation de classe est malgré tout bâtie sur le même principe que la dérivation « classique ».
- Les méthodes contiennent toutes le « type » concerné (dérivé ou non) en paramètre.
- T_RESERVATION_AFFAIRE déclarée après T_RESERVATION hérite des méthodes de celui-ci . Donc on pourra utiliser RESERVER et CHOIX_SIEGE avec un paramètre de type T_RESERVATION_AFFAIRE mais restreint aux champs valides (obtenu par conversion : voir la réalisation du body).
- De même T_RESERVATION_LUXE hérite des méthodes CHOIX_REPAS et de RESERVER de T_RESERVATION_AFFAIRE et de CHOIX_SIEGE par transitivité d'avec T_RESERVATION.
- Les procédures RESERVER devront être **redéfinies** pour les catégories «affaire » et «luxe » pour être utilisées (voir le body).

Imaginons maintenant l'ajout de la catégorie SUPERSONIQUE. Nous n'allons pas grossir le paquetage initial d'ailleurs assez chargé comme cela. Mais dans le style de l'extension de fonctionnalités nous allons créer un nouveau paquetage. Comme il n'y a pas de préfixage nous n'utilisons pas le concept de Père.Fils (on aurait pu le faire et cela sera fait plus loin). Ici on cherche avant tout à **étendre la structure** et par transitivité aussi les fonctionnalités (mais pour le type étendu).

```
with P_RESERVATION_95;
package P_RESERVATION_SUPER is

   type T_AGREMENT is (SALON, BAR, MUSCU);
```

```
   type T_RESERVATION_SUPER is new T_RESERVATION_LUXE with
      record
           AGREMENT : T_AGREMENT;
      end record;
   procedure CHOIX_AGREMENT (R : in out T_RESERVATION_SUPER);
   procedure RESERVER (R : in out T_RESERVATION_SUPER); -- surcharge
end P_RESERVATION_SUPER;
```

Les mêmes remarques sur l'héritage s'appliquent. Ainsi les méthodes : RESERVER, CHOIX_SIEGE, CHOIX_REPAS et CHOIX_VOITURE peuvent être utilisées avec un paramètre T_RESERVATION_SUPER (après conversion appropriée c'est à dire dans le sens enfant⇒ parent jamais dans l'autre sens évident !).

Et les body ?

D'abord (pour réviser un peu) le codage (compliqué !) des **case** de P_RESERVATION_83 :

```
package body P_RESERVATION_83 is
   procedure RESERVER (R : in out T_RESERVATION) is
   begin
      CHOIX_SIEGE (R);
      .......
      R.NUMERO_VOL := ...;
      R.DATE_VOYAGE := .....;
      R.NUM_SIEGE := ....;
      case R.CAT is
         when TOURISME => null;
         when AFFAIRE | LUXE => CHOIX_REPAS (R);
            case R.CAT is
               when TOURISME | AFFAIRE => null;
               when LUXE => CHOIX_VOITURE (R);
            end case;
      end case;
   end RESERVER;

   procedure CHOIX_SIEGE (R : in out T_RESERVATION) is
   begin
      ....
   end CHOIX_SIEGE;
   procedure CHOIX_REPAS (R : in out T_RESERVATION) is
   begin
      ....
   end CHOIX_REPAS;
   procedure CHOIX_VOITURE (R : in out T_RESERVATION) is
   begin
      ....
   end CHOIX_VOITURE;
   ........
end P_RESERVATION_83;
```

Le codage « suit » à la lettre les contours de la structure à initialiser (voir les spécifications) ! Lourd ! Et maintenant voyons le codage de la réalisation des spécifications avec la version à objets Ada95.

```
package body P_RESERVATION_95 is
   procedure RESERVER (R : in out T_RESERVATION) is
   begin
      CHOIX_SIEGE (R);
      .......
      R.NUMERO_VOL := ...;
      R.DATE_VOYAGE := .....;
      R.NUM_SIEGE := ....;
   end RESERVER;
```

```
   procedure RESERVER (R : in out T_RESERVATION_AFFAIRE) is
   begin
      RESERVER (T_RESERVATION(R));-- conversion sinon récursivité!
      CHOIX_REPAS (R);
   end RESERVER;
   procedure RESERVER (R : in out T_RESERVATION_LUXE) is
   begin
      RESERVER (T_RESERVATION_AFFAIRE(R));
      CHOIX_VOITURE (R);
   end RESERVER;
   procedure CHOIX_SIEGE (R : in out T_RESERVATION) is
   begin
      ....
      R.SIEGE := ......;
   end CHOIX_SIEGE;
   procedure CHOIX_REPAS (R : in out T_RESERVATION_AFFAIRE) is
   begin
      ....
      R.REPAS := ......;
   end CHOIX_REPAS;
   procedure CHOIX_VOITURE (R : in out T_RESERVATION_LUXE) is
   begin
      ....
      R.VOITURE := .....;
   end CHOIX_VOITURE;
end P_RESERVATION_95;
```

On peut suivre la composition **incrémentale** qui reprend **point par point** les structures **héritées** et n'ajoutant que ce qui est nécessaire **au fur et à mesure** et en n'utilisant que ce qui est déjà réalisé.

Les types étiquetés privés.

Il n'y a aucune difficulté à comprendre ce concept ! Pour réaliser le « privé » il suffit d'utiliser le mot réservé **private** et évidemment il faut rejeter les déclarations dans la partie « cachée ». Par exemple :

```
with P_RESERVATION_95;
package P_RESERVATION_SUPER is
   type T_RESERVATION_SUPER is new T_RESERVATION_LUXE with private;
   procedure CHOIX_AGREMENT (R : in out T_RESERVATION_SUPER);
   procedure RESERVER (R : in out T_RESERVATION_SUPER);
private
   type T_AGREMENT is (SALON, BAR, MUSCU);
   type T_RESERVATION_SUPER is new T_RESERVATION_LUXE with
      record
           AGREMENT : T_AGREMENT;
      end record;
end P_RESERVATION_SUPER;
```

De même et facilement le corps s'écrit comme pour les autres :

```
package body P_RESERVATION_SUPER is
   procedure RESERVER (R : in out T_RESERVATION_SUPER) is
   begin
      RESERVER (T_RESERVATION_LUXE(R));
      CHOIX_AGREMENT (R);
   end RESERVER;
```

```
   procedure CHOIX_AGREMENT (R : in out T_RESERVATION_SUPER) is
   .......
   end CHOIX_AGREMENT;
end P_RESERVATION_SUPER;
```

La programmation à échelle de classe.

Il est parfois intéressant de pouvoir faire référence, non pas à un type étiqueté précis (dérivé ou racine), mais à **l'ensemble des types** dérivés d'un type `tagged` (classe ou sous classe). Cette possibilité est obtenue à l'aide de l'attribut `'CLASS` que l'on utilise **avec un opérande qui est un type étiqueté** (ou dérivé) et non pas sur une instance de type (comme par exemple on pouvait utiliser l'attribut `RANGE` sur des instances de type tableau).

Prenons ceci (mais on aura une utilisation plus intelligente plus loin avec les paramètres) :

```
LE_TOURISTE : T_RESERVATION_TOURISME;
LE_COMMERCIAL : T_RESERVATION_AFFAIRE;
LE_PARVENU : T_RESERVATION_LUXE;
```

ces instanciations utilisent ici une notation dite de « **type spécifique** ». Mais par contre la déclaration :

Attention ! il faut initialiser à l'instar des types non contraints pour le compilateur !

```
L_INCONNU : T_RESERVATION'CLASS ……;
```

utilise la notation dite de « **type à échelle de classe** ».

L'objet instancié `L_INCONNU` pourra appartenir : soit au type « racine » `T_RESERVATION` ou aux quatre autres types dérivés `T_RESERVATION_AFFAIRE`, …., `T_RESERVATION_SUPER`.

Et avec : `LE_PRIVILÉGIÉ : T_RESERVATION_LUXE'CLASS ……;`
l'objet pourra appartenir seulement aux deux types `T_RESERVATION_LUXE` ou `T_RESERVATION_SUPER`.

L'intérêt du type à échelle de classe n'est donc intéressante qu'avec le typage des **paramètres formels** de sous programmes. On retrouve l'esprit des paramètres tableaux non contraints paramètres. Ainsi :

```
procedure RESERVER_UNE_PLACE (POUR : in out T_RESERVATION'CLASS) is
begin
   .......
   RESERVER (POUR); -- POUR, peut être de tag inconnu à la compilation!
   .......
end RESERVER_UNE_PLACE;
```

L'idée est de retarder l'instanciation. On pourra utiliser cette procédure avec un paramètre effectif d'un type appartenant à toute la « famille » (type racine ou type dérivé). Ainsi :

```
.......
RESERVER_UNE_PLACE (UN_INDIVIDU);
.......
```

Suivant le type de l'objet `UN_INDIVIDU` le corps de la procédure `RESERVER_UNE_PLACE` utilisera la procédure `RESERVER` dédié au type associé. Cette utilisation **peut même être retardée jusqu'au moment de l'exécution** du code si le type du paramètre effectif n'est reconnu qu'à ce moment là (notamment avec le type accès). Soit (on reverra cela au cours n°12) :

```
.......
RESERVER_UNE_PLACE (PTR_INDIVIDU.all);
.......
```

où `PTR_INDIVIDU` est de type `T_PTR_RESERVATION` déclaré lui même :

```
type T_PTR_RESERVATION is access all T_RESERVATION'CLASS;
```

avec cette illustration de type à échelle de classe nous venons de mettre en évidence les notions de **polymorphisme** et de **liaison dynamique**. Notions chères au « paradigme objet ». Et notions qui manquaient à Ada83 pour avoir le label (brevet) de **langage permettant les objets**. Remarquons aussi que Ada95 permet les objets et toutes les notions associées **même sans utiliser de pointeurs** ; ce qui n'est pas le cas de tous les langages à objets !

Vocabulaire (résumé *à lire et à relire*) :

L'extension de type est la propriété, à partir d'un type «parent » (étiqueté en Ada), **d'ajouter des champs** supplémentaires **quand on dérive** vers un type « enfant ». Les opérations héritées du type parent pour le type enfant ne sont **opérationnelles que sur leurs champs communs**. Elles **peuvent être redéfinies**. Des opérations (méthodes) supplémentaires **doivent être ajoutées pour les nouveaux champs**.

Le **polymorphisme** est la propriété d'un objet informatique **d'avoir des représentations multiples**. On a déjà vu ce concept avec le type article à discriminant (mutant ou non). Dans l'exemple précédent on voit un sous programme `RESERVER_UNE_PLACE` qui possède des paramètres se **référant à la classe d'un type étiqueté (ou dérivé)**. Le type de l'objet formel est **inconnu** et peut prendre des « formes » diverses (grâce au tag) adaptables au type du paramètre effectif.

La **liaison dynamique** (ou **ligature dynamique** chez certains auteurs) est justement le **mécanisme** mis en œuvre à **l'exécution** pour utiliser le sous programme en l'adaptant aux types des paramètres effectifs. On peut voir cela comme une **surcharge de la méthode qui est résolue à l'exécution**.

L'héritage multiple est la propriété de pouvoir dériver un type de plusieurs types à la fois. Il n'existe pas de construction syntaxique pour cette propriété en Ada. Mais cet objectif peut être réalisé par exemple avec des constructions de base du langage via la composition des génériques. A approfondir !

Remarque : Il n'y a pas en Ada de construction syntaxique proprement dite pour construire une classe comme cela existe dans certains langages (C++, Java, etc.). C'est sûrement ce qui gène les fanatiques de tel ou tel langage à objets ne retrouvant pas dans Ada leurs opérateurs et constructeurs traditionnels. Rappelons en résumé qu'une classe en Ada se conçoit avec la déclaration (dans un paquetage) d'un type étiqueté et de ses méthodes associées. Toute instanciation de ce type crée un objet. Toute dérivation du type étiqueté construit une sous classe qui hérite de la précédente. La dérivation peut avoir lieu dans le paquetage de la classe ou en dehors. Cette façon de faire (pour archaïque qu'elle puisse paraître) a été choisie pour conserver l'essentiel de ce qui était satisfaisant en Ada83 c'est-à-dire les merveilleux paquetages ! Voir en complément sur le CDRom la traduction d'un papier « Ada et les objets ».

Les types abstraits (`abstract`).

Ne pas confondre cette notion avec le T.A.D. (type « abstrait » **de données**) vue auparavant.
Un type abstrait est un type **étiqueté** introduit avec le mot réservé **`abstract`** supplémentaire (**`abstract tagged`**) éventuellement aussi (en plus) privé (**`abstract tagged private`**). Il permet de garantir une spécification commune à l'ensemble des types dérivés (appelée « racine » dans nos exemples antérieurs). C'est un **modèle de type étiqueté**.

Un type abstrait **ne peut avoir d'instance**. Les **types dérivés du type abstrait** ont évidemment des instances (sauf s'ils sont eux mêmes abstraits ! ce qui est possible).

Un type abstrait peut aussi déclarer des méthodes (sous-programmes) abstraites. Elles n'ont **pas de réalisation** pour le type racine (puisque le type n'est pas concret). Elles sont à **réécrire** pour chaque type dérivé.

Le plus souvent le contenu (champs) du type abstrait (**`abstract`**) est minimal (voire souvent vide). Pour illustrer cette notion nous allons reprendre nos réservations depuis un type racine abstrait (et vide).

```
package P_RESERVATION_95 is
   type T_RESERVATION is abstract tagged null record; -- vide!
   type PTR_RESERVATION is access all T_RESERVATION'CLASS; -- éventuel
   procedure RESERVER (R : in out T_RESERVATION) is abstract;
end P_RESERVATION;
```

La séparation en petits paquetages n'est pas nécessaire mais elle nous permet d'illustrer **l'association** des classes avec les bibliothèques hiérarchiques vues dans le cours n°7 (paquetage Père-Fils).

```
package P_RESERVATION_95.VOLS_SUBSONIQUE is  -- paquetage fils
   type T_SITUATION is (COULOIR, FENETRE);
   type T_RESERVATION_TOURISME is new T_RESERVATION with -- private possible
     record
        NUMERO_VOL : T_NUM_VOL;
        DATE_VOYAGE : T_DATE;
        NUM_SIEGE : T_NUM_SIEGE;
        SIEGE : T_SITUATION;
     end record;

   procedure CHOIX_SIEGE (R : in out T_RESERVATION_TOURISME);
   procedure RESERVER (R : in out T_RESERVATION_TOURISME);

-- ******************************************
   type T_REPAS is (VEGETARIEN, VIANDE_BLANCHE, VIANDE_ROUGE);
   type T_RESERVATION_AFFAIRE is new T_RESERVATION_TOURISME with
     record
          REPAS : T_REPAS;
     end record;

   procedure CHOIX_REPAS (R : in out T_RESERVATION_AFFAIRE);
   procedure RESERVER (R : in out T_RESERVATION_AFFAIRE);

-- ******************************************
   type T_VOITURE is (RENAULT, PEUGEOT, CITROEN);
   type T_RESERVATION_LUXE is new T_RESERVATION_AFFAIRE with
     record
          VOITURE : T_VOITURE;
     end record;

   procedure CHOIX_VOITURE (R : in out T_RESERVATION_LUXE);
   procedure RESERVER (R : in out T_RESERVATION_LUXE);
end P_RESERVATION_95.VOLS_SUBSONIQUE;
```

etc.. On peut reprendre `T_RESERVATION_SUPER` dans la foulée ou dans un autre paquetage **petit fils**.

Les types contrôlés (CONTROLLED).

Ada définit, dans le paquetage `ADA.FINALIZATION`, un type **abstrait contrôlé** (et aussi un **abstrait limité contrôlé**) respectivement `CONTROLLED` et `LIMITED_CONTROLLED`. Le paquetage exporte aussi des primitives respectivement `INITIALIZE`, `ADJUST`, `FINALIZE` d'une part pour le type `CONTROLLED` et `INITIALIZE`, `FINALIZE` d'autre part pour le type `LIMITED_CONTROLLED`. Ces primitives qu'on peut réécrire quand on dérive l'un des types `CONTROLLED` ou `LIMITED_CONTROLED` correspondent à l'action à entreprendre à l'élaboration de la définition d'une instance (`INITIALIZE`), à l'action d'affectation (`ADJUST`) et enfin à l'action de « suppression » quand l'instance « termine » sa portée (`FINALIZE`).

Schématiquement on a une utilisation analogue à ceci :

```
with ADA.FINALIZATION; use ADA.FINALIZATION;
package P_...... is
   type T_... is new CONTROLLED with private;
   ........
private
   type T_...... is new CONTROLLED with record
      ......
   end record;
   procedure  INITIALIZE (OBJET : in out T_....);
   procedure  ADJUST (OBJET : in out T_....);
   procedure  FINALIZE (OBJET : in out T_....);
   ........
end P_......;
```

Remarque :
Les procédures INITIALIZE et FINALIZE permettent de réaliser (et évidemment de façon transparente) les concepts de « **constructeur** » et de « **destructeur** », notions chères aux tenants de la programmation objets. Ces notions seront vues bientôt et surtout au deuxième trimestre dans le module de structure de données avec le C++. Quant à ADJUST elle permet de redéfinir l'affectation (s'il en est besoin évidemment) voir ci dessous.

Un exemple (convaincant ?) : Vous avez défini un type T_Date composé au moins des quatre informations évidentes : voir l'un des premiers exercices suggérés en algorithmique. Cette date est donc composée d'un jour, d'un numéro de jour, d'un mois et d'une année. La structure idéale pour implémenter cela c'est évidemment le type article. Tout cela est évidemment (bis !) réalisé dans un paquetage avec plein de bons principes : encapsulation, masquage, etc. Mais un doute subsiste quant au contrôle de la qualité et de la justesse d'une date car vous voulez manipuler des dates fiables : pas de 31 avril (évident !), ni de Jeudi 9 juin 2000 (moins évident). Une méthode lourde peut consister à ajouter un champ supplémentaire (booléen par exemple) aux quatre champs précédents ce booléen est mis à vrai après tout contrôle de validité de la date et testé systématiquement avant toute opération sur la date et une exception est levée évidemment si le booléen est à faux. Une autre méthode consiste à rejeter une seule fois le contrôle dans une utilisation **fréquente** de la date et ceci de façon transparente par exemple au moment de chaque affectation. En s'appuyant sur Adjust évidemment. Voyons cela.

```
type T_Date is private ; -- jusque là rien de nouveau !
```
……des déclarations diverses dont l'exception Exc_Date_Incorrecte.
```
private
  type T_Date is new Controlled with record – T_Date est dérivé
      Jour : ... ;
      Num : ... ;
      Mois : ... ;
      Annee : ... ;
  end record ;
```

Dans les déclarations diverses citées ci dessus il **faut** déclarer les trois sous programmes associés à tout type Controlled (type abstract par définition) soit :

```
procedure Initialize (Date : in out T_Date) ;
procedure Adjust (Date : in out T_Date) ;
procedure Finalize (Date : in out T_Date) ;
```

Dans le corps du paquetage encapsulant le type T_Date on définira les sous programmes. Initialize et Finalize sans effet seront définis avec l'instruction null !! Quant à Adjust (celui qui nous intéresse !) il contiendra un bon contrôle de l'objet Date avec levée éventuelle de l'exception Exc_Date_Incorrecte. Le contenu de ce contrôle est connu nous n'y reviendrons pas. Notez que chaque affectation d'une date de type T_Date entraîne l'exécution de Adjust sans que le développeur ou l'utilisateur n'ait à s'en préoccuper. Le contrôle est total et partout (même et surtout quand on ajoutera du code pour maintenir l'application. Sécurité oblige même si cela peut paraître lourd (c'est vrai en temps d'exécution) mais c'est terriblement efficace.

Remarque : qui s'applique à l'exemple précédent mais qui ne nécessite pas forcément la dérivation d'un type `controlled`. Si l'on déclare un type avec un discriminant formel tel que :

```
type T_Date (<>) is private ;
```

La notation (<>) qui marque aussi (mais de façon abstraite) la notion de discriminant oblige toute instanciation de ce type (c'est-à-dire toute déclaration d'une variable de ce type) à définir le discriminant. Or ici le discriminant n'étant pas précisé le compilateur demande impérativement une affectation avec la déclaration de la variable. C'est la notion de constructeur **obligatoire** chère à certains langages à objet (C++ par exemple, pour Java c'est facultatif car il existe un constructeur implicite). Pour réaliser cela en Ada il suffit de déclarer et définir une fonction qui retourne une structure de type `T_Date`.

```
function Date (paramètres formels) return T_Date ;
```

Alors la déclaration suivante :

```
Une_Date : T_Date ;
```

qui était refusée par le compilateur message : « unconstrained subtype not allowed (need initialization) »

avec l'écriture :

```
Une_Date : T_Date := Date (paramètres effectifs);
```

C'est correct.

Exercice : (développé en cours si le temps le permet) :

On souhaite concevoir un paquetage de gestion de comptes bancaires permettant de déclarer des instances d'un type `T_COMPTE_COURANT` (en réels décimaux) avec quelques méthodes telles `DEPOSER` et `RETIRER`. Cette réalisation est classique en voici l'ossature :

```
package P_GESTION_COMPTE is
   type T_COMPTE_COURANT is private; -- à gérer
   type T_EURO is delta 0.01 digits 10;-- la monnaie en question
   procedure DEPOSER(UN_COMPTE: in out T_COMPTE_COURANT;VALEUR : in T_EURO);
   procedure RETIRER(UN_COMPTE: in out T_COMPTE_COURANT;VALEUR: out T_EURO);
   private
      type T_COMPTE_COURANT is record
         SOLDE : T_EURO := 0.0; -- vide a priori
      end record;
end P_GESTION_COMPTE;
```

On souhaite dans chaque programme utilisateur tenir à jour un fichier «trace » des opérations (« dépôt » ou « retrait »). Ce fichier doit être ouvert (`APPEND_FILE`) à la déclaration de la première instance et refermé quand s'achève la vie de la dernière instance ! Il suffit pour cela de reprendre le paquetage ci-dessus en dérivant `T_COMPTE_COURANT` de `LIMITED_CONTROLLED`. On peut alors écrire (à faire!) les deux procédures `INITIALIZE` et `FINALIZE` qui réaliseront automatiquement le contrat demandé. Finir le body du paquetage. A revoir après le cours fichiers n°11 qui suit.

Différence entre généricité et héritage.

J.P. Rosen (toujours le même !) dans ces notes techniques dresse une étude comparative intéressante entre ces deux approches **généricité et héritage** (toutes deux très prisées comme techniques de développement de composants réutilisables). Ceci complète « la réutilisation » décrite à la page 9.

On trouvera ces fiches techniques à l'adresse: http://perso.wanadoo.fr/adalog/qtall.htm
Le complément qui nous intéresse ici est le n°18. A voir !

Cours 11 Ada les fichiers (4 heures)

Thème : les fichiers (textes, séquentiels, directs et flot de données)

Cette partie de cours fait suite aux cours n° 5 bis (Entrées-Sorties simples) et n° 9 (E/S « suite ») illustrant les « génériques ». Dans ces deux cours « préliminaires » on ne s'était intéressé qu'aux instructions de `TEXT_IO` permettant des `GET` et des `PUT` (clavier et écran) sur des types prédéfinis « classiques » (cours 5 bis) puis sur les **types construits scalaires** ou numériques prédéfinis (cours 9). Nous allons passer maintenant aux E/S fichiers. **L'annexe A du manuel de référence** traitant des fichiers et des entrées-sorties est très complète (**à éditer** !) Il y sera fait de nombreux renvois aux chapitres concernés (de A6 à A13). Les 4 heures de cours sont faites en deux fois 2 heures entrecoupées d'un autre cours de 2 heures (voir le planning au début du cours n°1).

Le fichier.

Un fichier est une structure complexe mais **organisée** permettant de **stocker** et de **restituer** des informations. On peut évidemment « `ECRIRE` » et « `LIRE` » dans cette structure. On distingue plusieurs classes de fichiers (voir plus loin) suivant les informations à stocker et les moyens d'accéder à ces informations. Le support matériel de nos jours est en général le disque dur (ou la disquette (en disparition) ou le CD-ROM).

L'information dans les fichiers est représentée :

- soit sous la forme «Latin_1» (ASCII étendu mais normalisé ISO) pas toujours bien reconnue par les terminaux (clavier et écran) comme on l'a remarqué. Cette forme est « visible ». **Ce sont les fichiers textes**. Voir tout le chapitre A10,
- soit sous la forme **code binaire** (suite d'octets). Le contenu du fichier n'est alors pas directement éditable à l'écran. Il faut utiliser un **logiciel** (en interface) pour l'interpréter.

La coutume veut que les **fichiers textes** soient souvent d'extension `.TXT` (sous `DOS` par exemple) ou sans extension. Ils sont « portables » sur tout matériel mais peu esthétiques (pas de format, de polices, de style etc.). Le contenu actuel des lettres électroniques (E-mail) appartiennent à cette catégorie et encore : il est fort recommandé de ne pas trop y mettre d'accent (pauvres européens continentaux !). Les fichiers de « commande » (ou script) appartiennent aussi à cette catégorie (`.BAT` en `DOS`). J'en oublie certainement d'autres.

Les autres fichiers (**non textes**) sont **multiples et variés**. On trouve dans cette catégorie les traditionnels fichiers « bureautiques » (les `.DOC` et les `.RTF` par exemple) où se mêlent caractères « éditables » (nommés graphiques en Latin_1) et caractères « non visibles » (nommés de contrôle en Latin_1) ayant une fonction particulière de structuration des informations. On trouve surtout dans cette catégorie les, non moins fameux, fichiers « binaires exécutables » (les `.COM`, les `.EXE`). Enfin on trouve les fichiers des bases de données, et/ou les fichiers mémorisant des types articles ou les fichiers « stream » vus plus loin et récemment les fichiers PDF (portables).

Remarque : les fichiers `HTML` (**H**yper **T**exte **M**arkup **L**anguage) de Internet sont inclassables. En effet s'ils sont bien des suite de caractères visibles «portables» ils comportent des «fonctions» à interpréter (entre les marqueurs (ou tag) représentés par les symboles < et >) mais .... visibles quand même.

Il existe deux modes **d'organisation** des fichiers (rien à voir avec « l'information stockée » vue ci-dessus) :

- le fichier à **organisation séquentielle** où tous les éléments sont accessibles l'un après l'autre à partir du premier (dans le style du ruban de la machine `MOUCHERON`). Les fichiers textes sont de cette famille mais ils ne sont pas les seuls; il existe aussi des « non textes » dans cette famille. Cette organisation est analogue à la notion de liste (notion vue dans le cours de Structure de Données, c'est pour bientôt).

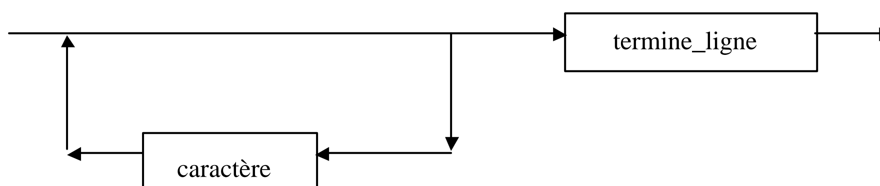
- le fichier à **organisation accès direct** où tous les éléments sont **directement** accessibles par leur rang c'est à dire à leur positionnement par rapport au premier élément qui a pour rang la valeur 1 (en Ada). En première approximation on retrouve dans ces fichiers la notion d'accès sélectif comme dans un vecteur.

Les fichiers textes Ada (tout le chapitre A10 du manuel de référence).

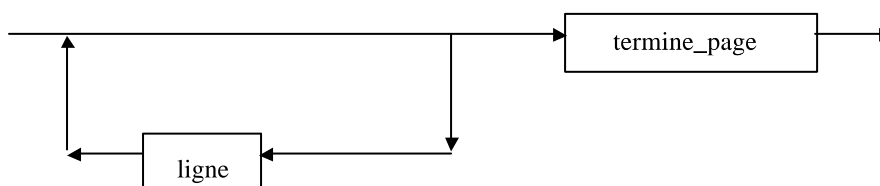
Définition - structure.

Un fichier texte est un objet structuré. Du point de vue logique, un fichier texte est une suite de caractères, de lignes, et éventuellement de pages. Une ligne est une "**liste**" (éventuellement vide) de caractères terminée par la *marque* "**termine_ligne**". Une page est une liste de lignes (éventuellement vide) terminée par la *marque* "**termine_page**" et un fichier est une '**liste**" de pages (éventuellement vides) terminées par une *marque* "**termine_fichier**". Attention une *marque* n'est pas forcément un caractère, ni un « unique » caractère spécial !

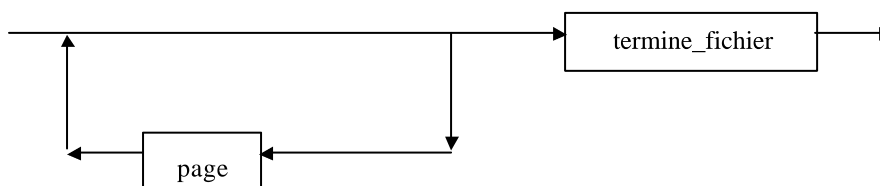
L'organisation peut être modélisée ainsi (notation « analogue » au D.S. mais ceci n'en est pas un !):
ligne :



page :



fichier texte :



"caractère" désigne l'un quelconque des caractères éditables (lettres, chiffres, caractères spéciaux tels que ponctuation, symboles opérateurs (* / + % etc.). La nature réelle des terminaisons **"termine_ligne"**, **"termine_page"** et **"termine_fichier"** **n'est pas définie par la norme du langage**. Mais ces terminaisons sont « en général » implémentées comme des caractères ou des suites de caractères, non éditables (dits de contrôle). Par exemple (en `Windows`) **termine_ligne** est « en général » la succession des deux caractères (`LATIN_1.CR` et `LATIN_1.LF`). En `UNIX` c'est uniquement `LATIN_1.LF`[1]. **Termine_fichier** est respectivement `CTRL-Z` ou `CTRL-D` (en `DOS` ou en `UNIX`). Cette distinction illustre la non compatibilité (légère) entre un fichier texte `DOS` ou `Windows` et `UNIX` ou `LINUX`. La structuration en page est assez peu usitée (mais possible !).

En Ada les caractères d'une ligne sont numérotés à partir de un. Il en est de même pour les lignes d'une page et les pages d'un fichier. Le numéro courant de colonne (numéro de caractère d'une ligne) est le numéro de colonne du prochain caractère à traiter. Voir chapitre A10 les deux premières pages. On reparlera plus loin de la procédure `SET_COL` **normalement utilisée pour éditer** mais **pratique pour saisir** des informations au clavier. Les numéros sont des valeurs appartenant au sous-type `POSITIVE_COUNT` du type de base `COUNT` (déclaré dans `ADA.TEXT_IO` voir en A.10.10), ainsi :

```
type COUNT is range  0..   ;   --  défini par l'implémentation
subtype POSITIVE_COUNT is range   1..COUNT'LAST ;
```

[1] Ce qui pose parfois des problèmes de portabilité (connus !).

Désignation et déclaration d'un fichier texte (voir le paquetage Ada.Text_Io chapitre A.10.1)

Les fichiers textes « manipulés » en Ada sont des objets du type limité privé **FILE_TYPE** exporté du paquetage ADA.TEXT_IO. Il est indispensable d'évoquer ce paquetage avec **with** évidemment. **use** éventuel !

Exemples (déclaration de 3 fichiers) **:**

```
F_ELEVES   : ADA.TEXT_IO.FILE_TYPE ; -- F_ELEVES : FILE_TYPE ; si use
F_EQUIPES  : ADA.TEXT_IO.FILE_TYPE ;
RUBAN      : ADA.TEXT_IO.FILE_TYPE ;
```

En Ada un fichier (texte ou non) est caractérisé par :

- un identificateur **interne** valable le « temps » de l'algorithme (**défini dans le programme**).
- un identificateur **externe connu du système** et **permanent** (sauf surnommage).

Exemples :
F_ELEVES, F_EQUIPES, et RUBAN déclarés ci-dessus sont des identificateurs **internes**.
"TS_MOUCHERON1.ADB", "lpr" (ou "PRN" imprimante sous DOS !) sont des identificateurs **externes**.

Mode d'utilisation (à ne pas confondre avec mode d'organisation !)

Les modes d'utilisation (voir A.7.10) autorisés pour les fichiers textes sont les modes IN_FILE, APPEND_FILE et OUT_FILE. Le mode IN_FILE limite l'utilisation du fichier aux seules **consultations** (lectures), le mode OUT_FILE correspond à l'autorisation **d'écriture** et APPEND_FILE permet **d'ajouter**, des informations, à la fin du fichier. Un fichier texte peut donc être soit lu (consulté), soit créé ou encore complété. La **modification** d'un fichier texte nécessitera une « recopie » : donc un algorithme à programmer (fait en TD-TP). Une trame est donnée en fin de ce cours 11 avant quelques exercices.

Ouverture d'un fichier texte (voir A.8.2 : file management).

« Ouvrir » un fichier est une instruction qui **précède** dans l'algorithme **l'utilisation** du fichier en question. Pour cela on dispose dans le paquetage TEXT_IO des deux procédures :

```
procedure CREATE    FILE  :  in out  FILE_TYPE ;
                    MODE  :  in   FILE_MODE  :=  OUT_FILE ;
                    NAME  :  in   STRING := "" ;
                    FORM  :  in   STRING := "") ;

procedure OPEN      (FILE :  in out  FILE_TYPE ;
                     MODE :  in   FILE_MODE ;
                     NAME :  in   STRING ;
                     FORM :  in   STRING := "") ;
```

La procédure CREATE **ouvre** un fichier en vue de sa **création** (sic !) (MODE := OUT_FILE) mode écriture obligatoire (évident!) et la procédure OPEN ouvre un fichier qui **existe déjà** sur disque (MODE au choix) soit alors en lecture, soit en écriture (mais l'ancien est perdu) ou en ajout. Si le fichier existe déjà l'ouverture avec CREATE le supprime ! Si le fichier n'existe pas l'ouverture avec OPEN lève une exception. Le fichier ne doit pas être déjà ouvert ! Ces deux procédures mériteraient d'être validées. Et même d'être réécrites. Voir en TD. Le «lien» (association) entre les noms de fichier (interne et externe) est fait grâce à CREATE ou OPEN.

Exemples :

```
CREATE    (F_EQUIPES, NAME => "FICH_OUT") ;
OPEN      (RUBAN, IN_FILE, "FICH_IN") ;
```

Le type FILE_MODE (énumératif) est défini dans TEXT_IO de la façon suivante :

```
type  FILE_MODE  is  (IN_FILE,  OUT_FILE, APPEND_FILE) ;
```

Une chaîne vide pour `FORM` spécifie l'utilisation des options par défaut de l'implémentation pour le fichier externe (à ne pas changer SVP ! Sauf connaissance très pointue du système d'exploitation).

Fermeture d'un fichier texte (voir A.8.2 : file management).

```
procedure  CLOSE (FILE : in out FILE_TYPE) ;
```

La procédure `CLOSE` **écrit** un **"termine_fichier"** (en mode `OUT_FILE ou APPEND_FILE`) et « **détruit** » le lien entre les deux noms de fichier (interne et externe) dans les trois modes. Le fichier doit être ouvert ? A voir !

Utilisation des longueurs de ligne et de page des fichiers textes.

Quand un fichier est ouvert **en sortie** (écriture ou ajout) (par exemple avec `CREATE`), les valeurs longueur maximale de lignes et longueur maximale de pages sont **mises à zéro** a priori. Attention, ce « zéro » signifie que les longueurs de lignes et de pages **ne sont pas bornées**. La constante `UNBOUNDED` est également fournie dans ce but pour réaliser cette action en cours d'algorithme.

1) Deux procédures permettent de fixer une limite au nombre de caractères d'une ligne ou du nombre de lignes d'une page.

```
procedure SET_LINE_LENGTH(FILE : in FILE_TYPE; TO : in COUNT);
procedure SET_PAGE_LENGTH(FILE : in FILE_TYPE; TO : in COUNT);
```

Attention l'effet est **non permanent** (Voir TD - TP), **il faut réinitialiser** ces valeurs à chaque changement de lignes ou de pages.

Exemples :

```
SET_LINE_LENGTH (F_TEXT, 80);
SET_PAGE_LENGTH (F_TEXT, UNBOUNDED) ;
SET_PAGE_LENGTH (F_ELEVES, 64) ;
```

2) Les fonctions `LINE_LENGTH` et `PAGE_LENGTH` permettent de **connaître**, à tout moment, la longueur maximale d'une ligne et la longueur maximale d'une page, d'un fichier ouvert en mode `IN_FILE` (évidemment).

```
function  LINE_LENGTH(FILE : in FILE_TYPE) return COUNT;
function  PAGE_LENGTH(FILE : in FILE_TYPE) return COUNT;
```

Contrôle des colonnes, lignes et pages (voir A.10.5)

1) Pour des fichiers ouverts en mode `IN_FILE` des fonctions permettent de tester si on se trouve :

- en fin de fichier :
```
function END_OF_FILE(FILE : in FILE_TYPE) return BOOLEAN;
```
- en fin de page :
```
function END_OF_PAGE(FILE : in FILE_TYPE) return BOOLEAN;
```
- en fin de ligne :
```
function END_OF_LINE(FILE : in FILE_TYPE) return BOOLEAN;
```

2) Des procédures permettent de sauter à la ligne ou à la page suivante (toujours`IN_FILE`).

- Saut de page en entrée :
```
procedure SKIP_PAGE(FILE : in FILE_TYPE);
```
- Saut de ligne en entrée :
```
procedure SKIP_LINE(FILE : in FILE_TYPE;
                    SPACING : in POSITIVE_COUNT:=1);
```

3) Pour des fichiers ouverts en mode `OUT_FILE` ou `APPEND_FILE` des fonctions permettent « d'écrire » :

- Un saut de ligne en sortie :
```
procedure NEW_LINE(FILE : in FILE_TYPE;
                   SPACING : in POSITIVE_COUNT:=1);
```

- Un saut de page en sortie :
```
    procedure NEW_PAGE(FILE : in FILE_TYPE);
```
4) Pour des fichiers ouverts (mode quelconque) des fonctions permettent de connaître le numéro :
- de la page :
```
        function PAGE(FILE : in FILE_TYPE) return POSITIVE_CONT;
```
- de la ligne :
```
        function LINE(FILE : in FILE_TYPE) return POSITIVE_COUNT;
```
- de la colonne :
```
        function COL(FILE  : in FILE_TYPE) return POSITIVE_COUNT;
```
5) Pour des fichiers ouverts (mode quelconque) on peut se positionner **plus loin** (pas de retour arrière) sur :
- une ligne :
```
        procedure SET_LINE(FILE : in FILE_TYPE;TO : in POSITIVE_COUNT);
```
- une colonne :
```
        procedure SET_COL(FILE : in FILE_TYPE;TO : in  POSITIVE_COUNT);
```

Cette dernière SET_COL est intéressante quand elle est aussi utilisée avec écran et clavier. (voir page 7).

Entrées-Sorties de caractères (voir A.10.7).

On retrouve, toujours dans le paquetage ADA.TEXT_IO, nos « vieilles connaissances » GET et PUT déjà vues (cours 5 bis) mais cette fois il y a un paramètre de plus c'est ..... le fichier :
```
procedure GET(FILE : in FILE_TYPE; ITEM : out CHARACTER);

procedure PUT(FILE : in FILE_TYPE; ITEM : in  CHARACTER);
```

Les procédures GET et PUT tiennent à jour les numéros courants de page, de colonne et de ligne du fichier spécifié.

- Chaque « transfert » d'un caractère ajoute 1 au numéro courant de colonne.
- Chaque « sortie » d'un **"termine_ligne"** met le numéro courant de colonne à 1 et ajoute 1 au numéro courant de ligne.
- GET s'utilise sur un fichier de mode IN_FILE et PUT sur les deux autres modes. Evident !
- La procédure GET lit le caractère suivant dans le fichier. Elle saute tout **"termine_ligne"** et **"termine_page"** rencontrés. Elle tient à jour les numéros.
- L'exception END_ERROR est levée si l'on tente de sauter un **"termine_fichier"**.
- La procédure PUT écrit le caractère dans le fichier. Si le numéro courant de colonne dépasse la longueur de ligne (si elle est spécifiée) la procédure PUT a pour effet d'appeler NEW_LINE (avec un ajout de 1 au numéro de lignes) **puis** écrit le caractère.

Exercice : Observer le corps du paquetage MOUCHERON. Il y a de nombreux exemples (à éditer et à méditer!).

Dans la nouvelle norme on trouve aussi les procédures LOOK_AHEAD et GET_IMMEDIATE pour un fichier en IN_FILE évidemment. Ces sous programmes sont intéressants dans des application interactives. Respectivement pour : lire **en avance sans consommer** donc sans avancer les compteurs (LOOK_AHEAD) ou lire **effectivement en consommant** mais sans avancer les compteurs (GET_IMMEDIATE). Voir Barnes page 525.

Entrées-Sorties de chaînes (paramètre STRING) (voir A.10.7)**.**

On retrouve les « classiques » vus en cours 5 bis (souvenez vous que : _LINE ≡ STRING) :

```
    procedure GET(FILE : in FILE_TYPE; ITEM : out STRING);

    procedure PUT(FILE : in FILE_TYPE; ITEM : in  STRING);

    procedure GET_LINE(FILE : in FILE_TYPE;
                       ITEM : out STRING; LAST : out NATURAL);
```

```
procedure PUT_LINE(FILE : in FILE_TYPE; ITEM : in STRING);
```

- La procédure `GET` détermine le nombre de caractères de la chaîne spécifiée `ITEM'LENGTH` et tente d'effectuer ce même nombre d'opérations `GET` pour des caractères successifs de la chaîne. Les **"termine_ligne"** et **"termine_page"** sont ignorés.
- La procédure `GET_LINE` remplace les caractères successifs de la chaîne spécifiée par les caractères successifs lus dans le fichier d'entrée. La lecture s'arrête si la fin de ligne est atteinte auquel cas la procédure `SKIP_LINE` est alors appelée avec un ajout de 1 au numéro de lignes. Le paramètre `LAST` est initialisé avec le nombre de caractères lus (de 0 à …..).
- La lecture s'arrête également si la chaîne est saturée. Dans ce cas la procédure `SKIP_LINE` n'est pas appelée donc le **termine_ligne** n'est pas sauté. Voir cours n°5 bis. Problème du clavier connu mais voir page 7 (bis !).
- La procédure `PUT` détermine le nombre de caractères de la chaîne et tente d'effectuer ce même nombre d'opérations `PUT` pour des caractères successifs de la chaîne.
- La procédure `PUT_LINE` appelle la procédure `PUT` pour la chaîne donnée puis la procédure `NEW_LINE` (avec un ajout de 1 au numéro de lignes).

Autres procédures et fonctions (voir A.8.2) (notez le **`in out`** ≡ fichier donnée-résultat avec les procédures**).**

```
procedure RESET(FILE : in out FILE_TYPE);
```

- Repositionne le fichier (ouvert avec `OPEN`) de telle sorte que la lecture de ces éléments puisse reprendre à partir du **début** du fichier (peut éventuellement être utilisé pour l'écriture mais !)

```
procedure RESET(FILE : in out FILE_TYPE; MODE : in FILE_MODE);
```

- Dans ce cas le paramètre `MODE` est précisé, et le mode courant du fichier est positionné au mode donné (permet de changer de mode).
- Si le fichier a pour mode courant `OUT_FILE` (par exemple avec `CREATE`) `RESET` a pour effet un appel à `NEW_PAGE` suivi d'une écriture d'un **"termine_fichier"**. Si le nouveau mode est `OUT_FILE` les longueurs de ligne et de page ne sont pas bornées. Si le nouveau mode est `IN_FILE` le fichier précédent est .... relu !

```
procedure  DELETE(FILE : in out FILE_TYPE);
```

- Cette procédure détruit le fichier externe associé au fichier donné. Le fichier spécifié est fermé et le fichier externe cesse d'exister (utile pour les fichiers dits temporaires). A bien maîtriser.

```
function  IS_OPEN(FILE : in FILE_TYPE) return BOOLEAN;
```

- Cette fonction retourne `TRUE` si le fichier est ouvert (c'est-à-dire s'il est associé à un fichier externe) et retourne `FALSE` dans le cas contraire.

```
function NAME (FILE : in FILE_TYPE) return STRING;
```

- Cette fonction retourne une chaîne qui identifie le fichier externe associé au fichier donné.

Exceptions dans les Entrées-Sorties.

Les exceptions suivantes peuvent être levées par des opérations vues précédemment. Elles sont déclarées dans le paquetage `ADA.IO_EXCEPTIONS` et renommées dans `ADA.TEXT_IO`. (Voir A.13).

`STATUS_ERROR`
- tentative d'action sur un fichier non ouvert ou encore pas les droits sur le fichier .
- tentative d'ouverture d'un fichier déjà ouvert

`MODE_ERROR`
- tentative de lecture (ou de test) de fin de fichier sur un fichier en mode `OUT_FILE` ou `APPEND_FILE`
- tentative d'écriture sur un fichier en mode `IN_FILE`

NAME_ERROR
- mauvais nom de fichier externe dans une procédure CREATE ou OPEN

USE_ERROR
- tentative d'opération qui n'est pas possible pour des raisons qui dépendent des caractéristiques du fichier externe (CREATE sur un fichier qui existe déjà et dont la réécriture n'est pas autorisée, DELETE sur un fichier protégé).

DEVICE_ERROR
- une opération d'Entrée-Sortie ne peut-être menée à bien à cause du mauvais fonctionnement du système sous-jacent.

END_ERROR
- tentative de saut du **"termine_fichier"**

DATA_ERROR
- voir les E/S sur les génériques (numériques) cours 9 et plus loin.

LAYOUT_ERROR
- tentative de positionnement de numéros de colonne ou de ligne au delà des longueurs maximales spécifiées.
- tentative d'exécution de PUT avec trop de caractères dans une chaîne.

Fichiers clavier/écran (voir A.10.3).

Le **clavier** (en lecture) et **l'écran** (en écriture) jouent un rôle particulier. Ils sont **considérés comme des fichiers** textes (dits **standard**) d'identificateurs (respectivement) : STANDARD_INPUT et STANDARD_OUTPUT. Ces identificateurs sont, a priori, ceux des identificateurs par défaut (CURRENT_INPUT et CURRENT_OUTPUT). Un identificateur pris **par défaut** signifie que le nom du fichier est **omis** dans les procédures et les fonctions de lecture/écriture. Ainsi :
GET(CARCOU); est équivalent à GET(STANDARD_INPUT, CARCOU);
PUT(CARCOU); est équivalent à PUT(STANDARD_OUTPUT, CARCOU);
On notera que les notions des cours 5 bis et 9 n'étaient qu'un cas particulier de ce cours sur les fichiers !
On rappelle que <u>ceci n'est valable qu'à la condition</u> qu'aucun autre fichier n'ait été, au préalable, déjà fixé (avec SET_INPUT ou SET_OUTPUT comme fichier courant (CURRENT) voir plus bas ces deux procédures.

Remarques (la deuxième est d'importance !) :
- On ne peut pas ouvrir, fermer, repositionner, ou détruire les fichiers d'entrée et de sortie standards.
- La lecture est tamponnée (effectuée par l'intermédiaire d'une zone tampon) et de ce fait, la saisie au clavier doit être suivie d'une frappe de la touche « entrée ». Une purge s'impose, le plus souvent, après ! Voir à ce sujet le cours n°5 bis où nous avions réécrit la procédure GET_LINE qui demandait un SKIP_LINE supplémentaire quand le nombre de caractères saisis correspondait **exactement à la taille** de la chaîne à saisir. Et même la lecture d'un Unbounded_String avec la fonction Get_Line ne règle rien ! Une **astuce très pratique** pour ne pas surcharger cette procédure est de **faire précéder toute lecture clavier** c'est-à-dire GET_LINE(chaine,long) ; de l'instruction SET_COL(STANDARD_INPUT,1) ; En effet si nous sommes en colonne 1 c'est qu'un SKIP_LINE a été effectué et le SET_COL est sans effet, si nous ne sommes pas en colonne 1 c'est qu'un **termine_ligne** est resté dans le tampon le SET_COL est alors à faire et le SET_COL effectue lui-même le fameux SKIP_LINE en allant au début de la ligne suivante !

SET_INPUT(FILE : **in** FILE_TYPE);
Le fichier par **défaut** en lecture (dit CURRENT_INPUT) sera le fichier passé en paramètre effectif.
Exemple : SET_INPUT(LE_FICHIER);
Le fichier par défaut en lecture n'est plus le fichier d'entrée standard STANDARD_INPUT. Si l'on souhaite utiliser le clavier il faut absolument écrire : GET(Standard_Input, Carcou) ;

SET_OUTPUT(FILE : **in** FILE_TYPE);
Le fichier par **défaut** en écriture (dit CURRENT_OUTPUT) sera le fichier passé en paramètre effectif.
Exemple : SET_OUTPUT(STANDARD_OUTPUT);
Le fichier par défaut en écriture **redevient** le fichier de sortie standard STANDARD_OUTPUT. A utiliser après un premier SET_OUTPUT.

Entrées-Sorties pour les types entiers (signés ou modulaires). Voir 1.10.8.

On retrouve le thème du cours 9 sur les Entrées-Sorties illustrant les génériques mais cette fois évidemment avec le concept de fichier en plus. Les procédures suivantes sont définies (pour les entiers signés) dans le sous paquetage générique `INTEGER_IO` inclus dans le paquetage `ADA.TEXT_IO`. (voir à partir de A.1052) ou pour les entiers modulaires le sous paquetage générique `MODULAR_IO`.
Le sous paquetage générique `INTEGER_IO` doit être instancié pour le type entier approprié (indiqué par le paramètre formel `NUM` dans les procédures).

```
Package INT_IO is new ADA.TEXT_IO.INTEGER_IO(Type_entier_approprié);
use INT_IO ;
```

Les procédures suivantes sont alors utilisables :

```
procedure GET(FILE : in FILE_TYPE; ITEM : out NUM; WIDTH : in FIELD:=0);

procedure PUT(FILE : in FILE_TYPE; ITEM : in NUM;
              WIDTH : in FIELD := DEFAULT_WIDTH;
              BASE  : in NUMBER_BASE  := DEFAULT_BASE);
```

`FIELD` est la largeur de champ et `NUMBER_BASE` indique la base des sous-types entiers `NUM`. Les valeurs de base sont du sous-type entier `NUMBER_BASE` telles que :

```
          subtype NUMBER_BASE is INTEGER range 2..16;
```

Les valeurs par défaut sont déclarées de la façon suivante :

```
DEFAULT_WIDTH  :  FIELD := NUM'WIDTH;
DEFAULT_BASE   :  NUMBER_BASE  :=  10;
```

`NUM'WIDTH` est la taille nécessaire pour éditer (ou imprimer) les plus longs entiers (souvent les plus petits !) c'est-à-dire `NUM'FIRST`. Par exemple sur une machine 32 bits (cas du `GNAT` avec `LINUX`) le type `INTEGER` implique que `INTEGER'FIRST` vaut -147583648 donc `INTEGER'WIDTH` vaut 11.

La procédure `GET` agit de la façon suivante en fonction du **troisième** paramètre `WIDTH` (à zéro par défaut) :

- Si la valeur de `WIDTH` est égale à 0 (= non bornée) alors `GET` saute tout blanc, **"termine_ligne"**, **"termine_page"** puis lit un éventuel signe (plus ou moins) puis lit selon la syntaxe d'un littéral entier (qui peut-être un littéral basé). La lecture s'arrête sur un non chiffre. Cf. Horner ! Mais c'est le système qui le fait !
- Si la valeur de `WIDTH` est donnée différente de zéro alors exactement `WIDTH` caractères sont lus (ou moins de caractères si **"termine_ligne"** est rencontré). Tout blanc de tête sauté est compris dans le comptage.
- `GET` retourne dans `ITEM`, la valeur de type `NUM`.
- L'exception `DATA_ERROR` est levée si la suite lue n'a pas la syntaxe exigée ou si la valeur obtenue n'est pas du sous-type `NUM`.

La procédure `PUT` édite la valeur du paramètre `ITEM`, sous forme d'un littéral entier, précédée du signe moins pour une valeur négative. Si `BASE` est indiquée `PUT` utilise la syntaxe des littéraux basés, avec toute lettre en majuscule. Les exemples suivants portent sur un type entier avec le deuxième paramètre (dit de formatage) `WIDTH`. Remarquez la justification à droite. `b` signifie espace ! remarquez que quand `WIDTH` est trop petit le système **<u>passe outre</u>** et édite la valeur correcte. Finalement on a intérêt à fixer `WIDTH` à 1 systématiquement !

Exemples : (`PUT` seul, donc `WIDTH` = 11 (valeur par défaut supposée)) :

```
          PUT (3) ;   ≡    "bbbbbbbbbb3"
          PUT (-5);   ≡    "bbbbbbbbb-5"
```

Exemples : (`PUT` et `WIDTH` fixé)

```
          PUT(12567,5);                    ≡ "12567"
          PUT(-35,4);                      ≡ "b-35"
          PUT(45,10);                      ≡ "bbbbbbbb45"
          PUT(125_345,1);                  ≡ "125345"
          PUT(2#1111_1111#,BASE=>16);      ≡ "16#FF#"
          PUT(2#1111_1111#,4);             ≡ "b255"   car base 10 par défaut
```

Entrées-Sorties pour les types réels (flottants ou fixes ou décimaux). Voir A.10.9.

Comme pour le cours 9 on doit instancier suivant le cas les paquetages génériques `FLOAT_IO` , `FIXED_IO` ou `DECIMAL_IO` inclus dans `ADA.TEXT_IO` (voir à partir de A.10.62). Ceux-ci doivent être instanciés pour le type point flottant prédéfini ou approprié ou pour le type point fixe `DURATION` ou approprié ou encore pour le type décimal approprié.

Exemples :

```
type  T_REEL  is  digits  8 ; -- réels flottants
package  ES_REEL  is new  ADA.TEXT_IO.FLOAT_IO (T_REEL) ;
use  ES_REEL ;

type  T_FIXE  is delta  0.1  range  -1.0..1.0 ; -- réels fixes
package  ES_FIXE  is new  ADA.TEXT_IO.FIXED_IO (T_FIXE) ;
use  ES_FIXE ;

type  T_DECIMAL  is delta  0.01  digits 9 ;-- réels décimaux max 9999999.99
package  ES_DECIMAL  is new  ADA.TEXT_IO.DECIMAL_IO (T_DECIMAL) ;
use  ES_DECIMAL ;
```

Les valeurs de type dénommé `NUM` sont éditées sous forme de littéraux selon les formats (déjà vus!) :

	`FORE . AFT E EXP`	pour le type point flottant (5 parties)
et	`FORE . AFT`	pour le type point fixe et le type décimal (3 parties)

`FORE`	partie entière
`AFT`	partie fractionnaire
`EXP`	partie exposant
`.`	point décimal
`E`	marque la notation avec exposant

On retrouve les procédures (avec `FILE` en plus) :

```
procedure  GET(FILE      :  in  FILE_TYPE;
               ITEM      :  in  NUM;
               WIDTH     :  in  FIELD := 0);

procedure  PUT(FILE      :  in  FILE_TYPE;
               ITEM      :  in  NUM;
               FORE      :  in  FIELD := DEFAULT_FORE;
               AFT       :  in  FIELD := DEFAULT_AFT;
               EXP       :  in  FIELD := DEFAULT_EXP);
```

avec pour `FIXED_IO` et `DECIMAL_IO` les valeurs par défaut suivantes :

```
DEFAULT_FORE  :  FIELD  :=  NUM'FORE;
DEFAULT_AFT   :  FIELD  :=  NUM'AFT;
DEFAULT_EXP   :  FIELD  :=  0;
```

et pour `FLOAT_IO` les valeurs par défaut suivantes (notation scientifique) :

```
DEFAULT_FORE :  FIELD  := 2 ;
DEFAULT_AFT  :  FIELD := NUM'DIGITS-1 ;
DEFAULT_EXP  :  FIELD := 3 ;   <---  Attention !
```

La procédure GET agit de la façon suivante :

- Si la valeur de WIDTH est égale à 0. GET saute tout blanc, tout **"termine_ligne"** ou **"termine_page"** puis lit un éventuel signe (+ ou -) puis lit selon la syntaxe d'un littéral réel (qui peut être basé)
- Si WIDTH est différent de 0. Alors exactement WIDTH caractères sont lus ou moins de caractères si un **"termine_ligne"** est rencontré. Tout blanc de tête sauté est compris dans le comptage.
- GET retourne dans ITEM la valeur de type NUM
- L'exception DATA_ERROR est levée si la suite lue n'a pas la syntaxe exigée ou si la valeur obtenue n'est pas du sous type NUM.

La procédure PUT :

- édite la valeur ITEM sous forme d'un littéral décimal avec le format défini par FORE, AFT, et EXP.
- Si le paramètre effectif correspondant à EXP a pour valeur 0 il n'y a pas d'exposant à l'édition. la partie entière comprend autant de chiffres qu'il est nécessaire pour représenter la partie entière outrepassant FORE si nécessaire. Le nombre de chiffres de la partie fractionnaire est donné par AFT il y a troncature éventuellement mais pas d'arrondi (à savoir !).

Exemples sans exposant :

```
PUT (FICHIER,235.00,5,3,0)       ≡      "bb235.000"
PUT (FICHIER,-12345.098,1,2,0)   ≡      "-12345.09"
```

Si le paramètre effectif correspondant à EXP est différent de 0 il y a un exposant suivant le format :

```
FORE . AFT  E  EXP
```

exemple :

```
PUT (FICHIER,235.03,FORE=>5,AFT=>2) ;  ≡  "bb235.03 E 00"
PUT (FICHIER,-1.230500,2,6,3)          ≡  "-1.230500 E 00"
```

si les arguments FORE , AFT et EXP ne sont pas précisés l'impression d'un nombre réel (digits) s'effectue sous une forme normalisée du style FORE . AFT E EXP mais avec :

une partie entière FORE (taille 2 caractères : le signe – ou un espace puis le premier chiffre significatif)
une partie fractionnaire AFT (taille NUM'DIGITS-1)
une partie exposant EXP (taille 3 caractères)

Exemple : supposons : FLOAT'DIGITS = 9

```
PUT (FICHIER,123.5) ;        est formaté     "b1.23500000E+02"
PUT (FICHIER,-234.5) ;                       "-2.34500000E+01"
PUT (FICHIER,-0.356E7) ;                     "-3.56000000E+06"
PUT (FICHIER,0.000456) ;                     "b4.56000000E-04"
```

On verra d'autres exemples en cours et en TD-TP.

Entrées-Sorties pour les types énumératifs. Voir A.10.10.

On doit (cf. cours 9) instancier le sous paquetage générique ENUMERATION_IO de ADA.TEXT_IO avec le type énumératif approprié.

```
package ENUM_IO is new ADA.TEXT_IO.ENUMERATION_IO (Type_énumératif);
use  ENUM_IO ;
```

On retrouve les procédures :

```
     procedure GET(FILE : in FILE_TYPE;ITEM : out ENUM);

     procedure PUT(FILE :     in  FILE_TYPE ;
                   ITEM  :    in  ENUM ;
                   WIDTH :    in  FIELD := DEFAULT_WIDTH ;
                   SET   :    in  TYPE_SET := DEFAULT_SETTING) ;
```

avec

```
          type TYPE_SET  is  (LOWER_CASE, UPPER_CASE) ;
          DEFAULT_WIDTH   :  FIELD := 0 ;
          DEFAULT_SETTING :  TYPE_SET := UPPER_CASE ;
```

La procédure GET agit de la façon suivante :

- Après avoir sauté tout blanc, **"termine_ligne"**, **"termine_page"** GET lit un identificateur (majuscules ou minuscules) et retourne dans ITEM la valeur du type énumératif.
- L'exception DATA_ERROR est levée si la suite lue n'a pas la syntaxe voulue ou si l'identificateur ne correspond pas à une valeur du sous-type énumératif.

La procédure PUT :

- édite la valeur du paramètre ITEM sous forme d'un littéral d'énumération.
- Le paramètre facultatif SET indique si des majuscules ou des minuscules sont utilisées pour des identificateurs (a priori c'est en majuscule !).

Ce cours de 2 heures se poursuivra ensuite (2 autres heures) avec l 'étude des fichiers « non textes » c'est-à-dire SEQUENTIAL_IO, DIRECT_IO et STREAM_IO. Mais entre temps (la semaine prochaine) le cours sur la testabilité sera étudié où l'utilisation des fichiers textes est fort utile.

Compléments (sur le cours n°9 généricité) :

Pour effectuer des saisies de numériques (entiers ou réels) **valides** il est intéressant de redéfinir les GET de cette façon :

D'abord le type est déclaré :

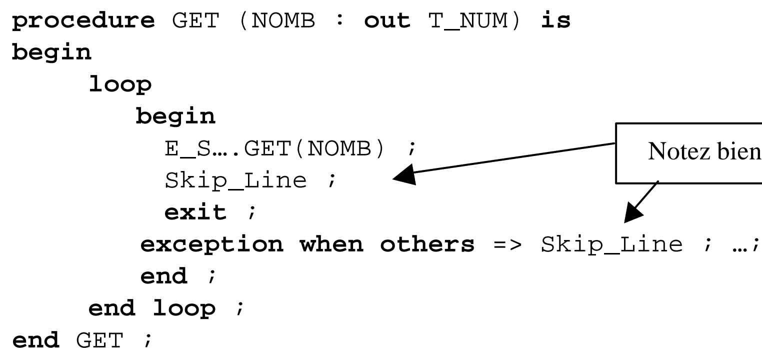
Choisis suivant le type

```
type T_NUM is ………(range, digits, delta) ;
```

Puis le sous paquetage ADA.TEXT_IO_….. approprié au type est instancié :

```
package E_S…. is new ADA.TEXT_IO_…..(T_NUM) ; -- pas de use
```

enfin on écrit la surcharge :



```
procedure GET (NOMB : out T_NUM) is
begin
     loop
        begin
          E_S….GET(NOMB) ;
          Skip_Line ;
          exit ;
        exception when others => Skip_Line ; …;
        end ;
     end loop ;
end GET ;
```

Notez bien ! les deux !

quant à la procédure `PUT` elle peut être aussi surchargée ainsi :

```
procedure PUT (NOMB : in T_NUM) is
begin
     E_S….PUT(NOMB, paramètres de formatage) ;
end PUT ;
```

Entrées-Sorties sur les `String` (non `Fixed`).

Les `Unbounded_String` :

On rappelle (vu en TD-TP 12) que `Ada.Strings.Unbounded.Text_Io` est le paquetage qui leur est dédié pour les entrées sorties.

On retrouve les traditionnelles procédure `Put` et `Put_Line` (avec ou sans fichier). Mais il faut noter que pour `Get_Line` il s'agit d'une fonction et non d'une procédure. Il n'y a pas de `Get` !

Les `Bounded_String` :

On rappelle (vu en TD-TP 12) qu'il n'existe pas de paquetage prévu (le type étant issu d'un paquetage générique). Il est cependant commode et facile de concevoir un tel paquetage très pratique. Fait en exercice avec en plus l'illustration d'un paquetage générique paramètre de généricité d'un autre paquetage ! On a voulu dans cet exercice d'entrées sorties de `Bounded_String` conserver les mêmes sous programmes que pour les `Unbounded_String`. Question d'homogénéïté !

Trame de parcours d'un fichier texte (à revoir et à creuser en TD) :

Exemple de parcours simplifié d'un fichier texte caractère par caractère avec édition écran :

```
with ADA.TEXT_IO ;use ADA.TEXT_IO ;
procedure  PARCOURS  is
     C  :  CHARACTER ;
     F_TEXTE  :  ADA.TEXT_IO.FILE_TYPE ;
begin
     OPEN(F_TEXTE, IN_FILE, "nom_externe "); -- plus dynamique SVP !
     loop
     exit when (END_OF_FILE(F_TEXTE));
          if (END_OF_PAGE(F_TEXTE))
          then
               SKIP_PAGE(F_TEXTE) ;
               NEW_PAGE;
          elsif (END_OF_LINE(F_TEXTE))
               then
               SKIP_LINE(F_TEXTE) ;
               NEW_LINE ;
          else
               GET(F_TEXTE, C) ; -- lecture du caractère
               PUT(C) ;          -- traitement !
          end if ;
     end loop ;
     CLOSE(F_TEXTE) ;
end  PARCOURS ;
```

Exercice 1 :
Ecrire une procédure permettant de copier un fichier (`F_TEXT1`) sur un autre fichier (`F_TEXT2`).

- caractère par caractère
- ligne à ligne

Exercice 2 :
Lire un fichier texte contenant une ou plusieurs valeurs entières basées (séparées par au moins un espace) par ligne en se servant des Entrées-Sorties pour les types entiers. On affichera à l'écran les valeurs et on créera un nouveau fichier contenant les mêmes valeurs mais en notation décimale (donc non basée).

Exercice 3 :
Reprendre l'exercice précédent mais en se servant des Entrées-Sorties pour les types « flottants » puis pour les types point fixe puis pour les types décimaux.

Exercice 4 :
Lire un fichier texte contenant une ou plusieurs valeurs énumératives par ligne en se servant des Entrées-Sorties pour les types énumératifs (par exemple `T_JOUR` ou `T_MOIS`)..

Exercice 5 (synthèse) :
Lire un fichier texte qui contient sur chaque ligne une valeur de type `T_JOUR`, en entier, un réel et une chaîne de 10 caractères. Afficher à l'écran avec une pause à chaque édition.

Exercice 6 :
Ecrire des primitives `CREER` et `OUVRIR` qui valide sérieusement le `CREATE` et le `OPEN`.

Exercice 7 :
Examiner les réalisations (body) du paquetage `P_MOUCHERON` (beaux exemples !).

Les fichiers « séquentiels ». Voir aussi le manuel de référence en A.8 (polycopié paquetages)**.**
Ce cours (2 heures) pourrait être présenté selon les circonstances après le cours n°12 sur le type `access`.

Définition et structure.

En accès séquentiel le fichier est vu comme **une suite de valeurs de même type** (donc pas seulement le type caractère comme pour un fichier texte). Un fichier texte est un cas « très particulier » de fichier séquentiel. Ces valeurs typées sont «transférées » (mémoire ⇒ disque ou disque ⇒ mémoire) dans l'ordre où elles apparaissent. Le type géré dans les fichiers séquentiels est souvent un type **article**, ce qui explique l'appellation traditionnelle de « **enregistrement** » à l'entité lue ou écrite dans de tels fichiers (tradition aussi venue du `COBOL` langage des années 1960 dédié à la gestion [2]). Le fichier étant ouvert, le « transfert » commence toujours à partir du début du fichier (en lecture comme en écriture mais **pas en ajout** évidemment). Seul le concept de fin de fichier (`END_OF_FILE`) reste présent. Les « fin de lignes, fin de pages et la notion de colonne » ont disparu ! Pour ces fichiers séquentiels les Entrées-Sorties de valeurs (d'un même type d'élément) sont définies au moyen du paquetage générique `SEQUENTIAL_IO`. (attention il ne s'agit plus de `TEXT_IO` !). Et le paramètre générique peut être même à discriminant voir en A.8 la déclaration du paquetage qui commence par :

```
generic
     type Element_Type (<>) is private ; -- générique même non contraint !
package Ada.Sequential_Io is
```

Pour définir les Entrées-Sorties séquentielles pour un type d'élément concerné, on doit, comme toujours, déclarer une instanciation de cette unité générique, avec le type donné comme paramètre générique effectif.

Exemple :

```
package  SEQ_IO  is new  SEQUENTIAL_IO  (T_ARTICLE);
use    SEQ_IO ;
```

Chaque instanciation du paquetage génétique `SEQUENTIAL_IO` déclare donc un type `FILE_TYPE` **différent**. Remarquez que dans le cas de `TEXT_IO` (cours 5 bis et même en 9 bis) le type `FILE_TYPE` **était unique** et donc commun aux différents type de données (caractères, chaînes, entier (signés ou modulaires), réels (fixes ou flottants ou encore énumératifs) !

Gestion des fichiers « séquentiels ».

Mode d'utilisation.

Les modes autorisés pour les fichiers séquentiels sont les modes `IN_FILE`, `APPEND_FILE` et `OUT_FILE` (comme pour `TEXT_IO` et c'est cohérent puisque s'agissait « en gros » d'un fichier séquentiel de caractères!).
La fonction `MODE` retourne le mode courant du fichier donné :

```
     function  MODE (FILE : in FILE_TYPE) return FILE_MODE;
```

Désignation et déclaration d'un fichier.

Les fichiers séquentiels Ada sont des objets d'un type `FILE_TYPE` déclaré par instanciation de `SEQUENTIAL_IO`. Donc les fichiers sont déclarés de la façon suivante :

Exemple :

```
     FICH_SEQ1 : SEQ_IO.FILE_TYPE ;
```

Ou encore mieux :

```
     subtype T_Fic_Seq is SEQ_IO.FILE_TYPE ;-- sous-type fichier séquentiel
     FICH_SEQ1 : T_Fic_Seq;
```

`FICH_SEQ1` est le nom interne du fichier. Le préfixage est recommandé à la déclaration. L'utilisation de sous type allégera vos déclarations (voir ci dessus).

---

[2] qui résiste encore ! « Ils y reviendront au Cobol !! (gag !)

Ouverture et fermeture d'un fichier séquentiel.

Les procédures CREATE, OPEN, CLOSE, DELETE, RESET et les fonctions END_OF_FILE, NAME, MODE et IS_OPEN déclarées par instanciation du paquetage SEQUENTIAL_IO sont identiques quant à leurs profils à leurs homologues de gestion de fichier texte. Plus de END_OF_LINE ! ni END_OF_PAGE ! Déjà dit.

Attention : la norme du langage Ada ne définit pas ce qui arrive aux fichiers externes après la fin de l'exécution du programme principal si les fichiers n'ont pas été fermés.

Entrées-Sorties séquentielles (attention c'est nouveau).

Les opérations disponibles sont (**attention il n'y a plus** de GET ni de PUT!) :

```
procedure  READ(FILE  :  in   FILE_TYPE ;
                ITEM  :  out  ELEMENT_TYPE) ;

procedure  WRITE(FILE  :  in  FILE_TYPE ;
                 ITEM  :  in  ELEMENT_TYPE) ;
```

La procédure READ agit sur le fichier en mode IN_FILE. Elle lit un élément du fichier et retourne la valeur de cet élément dans le paramètre résultat ITEM.

La procédure WRITE agit sur un fichier en mode OUT_FILE ou APPEND_FILE. Elle écrit la valeur de ITEM dans le fichier.

Dans les deux cas MODE_ERROR est levée si le mode est incorrect.

La fonction END_OF_FILE retourne TRUE s'il n'y a plus d'éléments qui puissent être lus.

```
function END_OF_FILE (FILE : in FILE_TYPE) return BOOLEAN ;
```

Les fichiers « directs ». Voir en A.8.4 le manuel de référence .

En accès direct le fichier est vu comme un ensemble d'éléments de même type occupant des positions consécutives en ordre linéaire. Une valeur peut être transférée de, ou vers, un élément du fichier situé à **n'importe quelle position choisie**. La position d'un élément est spécifiée par un indice (ou rang ou index) qui est un nombre de type COUNT**. Le premier élément a pour rang 1**.

Un fichier direct gère un **indice courant**. Cet indice courant est **utilisé par la prochaine opération** de lecture ou d'écriture. A l'ouverture l'indice courant est implicitement positionné à 1. Cet indice courant est la propriété de l'objet fichier interne. Il peut être **géré automatiquement** par les opérations d'entrées sorties mais aussi par **quelques instructions spéciales** mises en œuvre par le programmeur.

Pour ces fichiers les Entrées-Sorties de valeurs (d'un **même type d'élément**) sont définies au moyen du paquetage générique DIRECT_IO. Voir A.8.4 la déclaration du paquetage. Mais le paramètre générique n'est **pas à discriminant** comme cela était possible avec les fichiers séquentiels. APPEND_FILE a disparu (à revoir).

Pour définir les Entrées-Sorties directes pour un type d'élément donné, on doit déclarer une instanciation de cette unité générique avec le type concerné comme paramètre générique effectif.

Exemple :

```
package  DIR_IO  is new  ADA.DIRECT_IO  (type d'élément approprié);
use  DIR_IO ;
```

Chaque instanciation du paquetage générique DIRECT_IO déclare un type FILE_TYPE différent (comme pour les fichiers séquentiels vus précédemment). **Faites des sous types fichiers** !

Gestion des fichiers « directs ». Voir aussi en A.8.5.

Mode d'utilisation.
Les 3 (différents !) modes IN_FILE, INOUT_FILE et OUT_FILE sont autorisés pour les fichiers directs. Le mode INOUT_FILE permet des lectures et/ou des écritures, dans le même algorithme, et sur un même fichier. Le mode APPEND_FILE n'existe plus car il est avantageusement remplacé par INOUT_FILE !

Désignation et déclaration d'un fichier « direct »

Les fichiers directs en Ada sont des objets d'un type FILE_TYPE déclaré par instanciation de DIRECT_IO.
Ces fichiers sont déclarés de la façon suivante :

Exemple :
```
     FICH_DIR1  :  DIR_IO.FILE_TYPE ;
```
Ou mieux encore :
```
     subtype T_Fic_Dir is DIR_IO.FILE_TYPE ;-- sous-type fichier direct
     FICH_DIR1 : T_Fic_Dir;
```

FICH_DIR1 est le nom interne du fichier. Le préfixage est recommandé comme pour les séquentiels.

Ouverture et fermeture d'un fichier direct.

Les procédures CREATE, OPEN, CLOSE, DELETE, RESET et les fonctions END_OF_FILE, NAME, MODE et IS_OPEN déclarées par instanciation du paquetage générique DIRECT_IO sont identiques quant à leurs profils à leurs homologues de gestion de fichier séquentiel.

Entrées-Sorties directes (attention encore des nouveautés !)

Les procédures READ pour les fichiers ouverts en mode IN_FILE et INOUT_FILE sont :
```
     procedure  READ(FILE  :  in   FILE_TYPE ;
                     ITEM  :  out  ELEMENT_TYPE ;
                     FROM  :  in   POSITIVE_COUNT) ;
     procedure  READ(FILE  :  in   FILE_TYPE ;
                     ITEM  :  out  ELEMENT_TYPE) ;
```

- La première forme de l'instruction READ agit sur un fichier en mode IN_FILE ou INOUT_FILE tandis que la dernière agit sur un fichier en mode IN_FILE seulement.
- La procédure READ (première forme) positionne l'indice courant à la valeur donnée par le paramètre FROM puis retourne dans le paramètre ITEM la valeur de l'élément dont la position dans le fichier donné est spécifiée par l'indice courant. Cette procédure READ peut **alterner** avec WRITE dans l'algorithme !
- La procédure READ (deuxième forme) travaille avec la valeur courante de l'indice courant (qu'elle gère elle même) et retourne dans le paramètre ITEM la valeur de l'élément situé à cet endroit.

Les procédures WRITE pour les fichiers ouverts en mode INOUT_FILE et OUT_FILE sont :
```
     procedure  WRITE(FILE    :  in  FILE_TYPE ;
                      ITEM    :  in  ELEMENT_TYPE ;
                      TO      :  in  POSITIVE_COUNT) ;
     procedure  WRITE(FILE  :  in  FILE_TYPE ;
                      ITEM  :  in  ELEMENT_TYPE) ;
```

La procédure WRITE (première forme) positionne l'indice courant à la valeur donnée par le paramètre TO puis (pour les deux formes) « écrit », dans le fichier donné, la valeur du paramètre ITEM à la position spécifiée par l'indice courant (celui-ci est utilisé en l'état dans la deuxième forme).

Remarque très importante :
Chaque opération READ ou WRITE se termine en augmentant l'indice courant d'une position (prêt à lire ou à écrire l'enregistrement suivant). Donc attention à la séquence : lecture ⇒ modification ⇒ écriture (séquence possible sur un fichier en mode INOUT_FILE) car sans gestion de l'indice courant on écrira la modification sur l'enregistrement suivant et non sur l'enregistrement à modifier. Avant la réécriture (ou dans la même instruction) il faut **initialiser à nouveau** l'indice (il était prudent de le mémoriser) à la valeur de la lecture. De façon générale il est prudent de ne pas mêler Read ou Write à deux paramètres avec ceux à trois paramètres.
On dispose en outre des procédures et fonctions (de gestion de l'indice avec une variable de type COUNT):

```
procedure SET_INDEX(FILE : in FILE_TYPE; TO : in POSITIVE_COUNT);
function  INDEX(FILE : in FILE_TYPE) return POSITIVE_COUNT;
function  SIZE(FILE  : in FILE_TYPE) return COUNT;
```

La procédure SET_INDEX agir sur un fichier de mode quelconque. Elle positionne l'indice courant du fichier à la valeur de l'indice donnée par le paramètre TO. La fonction INDEX retourne la valeur de l'indice courant du fichier donné. Enfin la fonction SIZE agit sur un fichier de mode quelconque et retourne la taille courante du fichier externe qui est associé au fichier interne. **Attention il n'y a pas de trou** (au sens physique) dans un fichier direct : si un enregistrement n° N est créé avant celui de n° N-1 ce dernier N-1 existe **mais n'est pas initialisé** !

Les fichiers « Stream ». (voir le chapitre 13 et en A.12 dans le manuel de référence à éditer !).

Problème (attention c'est beau mais hard ! et nous ne verrons pas tout !). La traduction approximative de Stream peut être « flot de données ». Voyons le détail d'un tel concept. S'il est facile d'imaginer la manipulation, dans un fichier, de données **homogènes** donc tous de même type (entiers, réels et même articles) il en va autrement pour des données **hétérogènes** c'est-à-dire un patchwork de données arbitraires de types différents. Le « arbitraire » étant connu quand même de celui qui écrit (normal !) mais aussi de celui qui lit. Pour simplifier on pourra imaginer qu'un « magicien » va transformer en suite d'octets les variables ou valeurs d'un type particulier dans un sens et dans l'autre (lecture et écriture). Cette idée est très intéressante car on peut continuer à travailler (merci Ada) avec des vues abstraites d'un type de données qui seront traduites en paquets d'octets stockables dans un fichier (dit de flot) ou même envoyées sur un réseau (style ftp par exemple !). Le fichier contient des données **hétérogènes convertibles** « élégamment » **par le langage** d'un côté comme de l'autre. On retrouve encore tous les avantages de la haute sécurité Ada et notamment le fort typage et la portabilité.

Mise en œuvre. (paquetages Ada.Streams et Ada.Streams.Stream_Io).

Pour lire et écrire des flots il faut d'abord manipuler un type ; ce sera un type dérivé d'une classe abstraite, limitée et privée (rien que cela !). Quel bel exemple du cours n°10 ! Voir dans le paquetage Ada.Streams le type Root_Stream_Type. Puis il faut le type de l'implémentation (dans notre exemple l'octet) voir le type Stream_Element (type modulaire !) et bien sûr le type tableau (non contraint !) de tels éléments sans oublier le type fichier de flot (FILE_TYPE toujours !). Puis comme bouquet final les deux procédures Read et Write mais abstraites évidemment. Bref bel exemple mais … abstrait s'il en est. Dur, dur mais que c'est beau ! Heureusement il existe un paquetage fils dédié aux entrées et sorties c'est Ada.Streams.Stream_IO. Il ressemble comme un frère au paquetage SEQUENTIAL_IO (généricité en moins évidemment !). Il travaille avec le type Stream_Access (pointeur sur Root_Stream_Type'Class). Il permet de lire et d'écrire des suites de Stream_Element (pour simplifier pensez en terme d'octets). Mais qu'en est-il de la conversion des suites de Stream_Element en le type sous-jacent ? Sûrement pas à la portée du premier programmeur venu. Heureusement (bis) il existe des attributs 'Write et 'Read s'appliquant au type en question qui apparaît enfin « en clair ». Ces attributs, à partir du fichier de flot (via le pointeur), et du type en préfixe convertissent automatiquement dans un sens ou dans l'autre les suites de Stream_Element tout en réalisant les opérations de lecture et d'écriture. La seule précaution à prendre est d'opérer sur le nom du fichier interne une opération de redirection vers le type access mais réalisée grâce à la fonction Stream. Ouf ! Comme un bon dessin vaut mieux qu'un long discours on verra un exemple tiré du TP T_Appartement fait en TD-TP (semaine n° 12) puis en exercice de révision final à la fin de l'année TD-TP n° 19 (semaine 14).

Avant cet exemple voici un schéma grossier d'une telle utilisation (Barnes page 526 [3]) :

```
with Ada.Streams.Stream_IO ;
-- permet l'accès à la ressource Ada.Streams
-- avec entre autres le type Stream_Element
-- et l'accès au fils Stream_IO avec le type FILE_TYPE
-- et les opérations habituelles sur les fichiers :
-- (OPEN, CLOSE, End_Of_File, etc.) ainsi que
-- la fonction Stream qui à partir d'un fichier de flot retourne un access
…………………..
use Ada.Streams.Stream_IO ;
…………….
FICHIER_FLOT : FILE_TYPE ; -- un fichier de flot de Stream_IO
ACCES_FICH : Stream_Access ; -- son pointeur (voir cours n°12)

-- on suppose déclarés : le type T_DATE, les types T_ENTIER et T_REEL
…………..
begin
      CREATE (FICHIER_FLOT,..,..) ; -- création du fichier
      ACCES_FICH := Stream (FICHIER_FLOT) ; -- pointeur sur fichier
    loop………
      T_DATE'WRITE(ACCES_FICH,DEMAIN) ; --écriture d'une date
      T_ENTIER'WRITE(ACCES_FICH,L_ENTIER) ; -- d'un entier
      T_REEL'WRITE(ACCES_FICH,LE_REEL) ; -- d'un réel etc.
    ……….
Et ainsi de suite
     end loop ;………
      CLOSE(FICHIER_FLOT) ;
end ;
```

Notez les étapes importantes !

Le fichier (binaire) créé avec `CREATE` et avec des séquences d'écritures `WRITE` contient des suites de paquets de bits (sûrement des octets) interprétables un jour par un autre programme après évocation de `Ada.Streams.Stream_IO` et de l'attribut `'READ` opérant sur les types `T_DATE`, `T_ENTIER`, etc. **dans le même ordre** que l'écriture !

Remarque :
Ce n'est pas l'utilisateur qui fixe le codage binaire de stockage des valeurs (c'était possible en mémoire avec des pragmas) mais c'est le langage qui le fait de façon transparente. Comme Ada est normalisé et que les compilateurs respectent la norme **il n'y a aucun problème de portabilité** ! Le seul protocole à connaître entre les utilisateurs est l'ordre de stockage des données (c'est bien le moins !).

L'utilisateur peut même redéfinir les opérations d'écriture ou de lecture en écrivant une procédure telle :

```
procedure ECRIRE (Stream : access Root_Stream_Type'Class ;
                  VAL : in Le_type) ;
for Le_Type'WRITE use ECRIRE ; -- clause d'utilisation
```

avec la clause de représentation **`for`** l'attribut `'WRITE` utilisera `ECRIRE` ainsi défini.
Plus loin on définit la procédure :

```
procedure ECRIRE (Stream : access Root_Stream_Type'Class ;
                  VAL : in Le_type) ; is
begin
      ……. Ici le détail des écritures
end ECRIRE ;
```

[3] voir aussi pages suivantes 527-528.

Il existe aussi d'autres fonctionnalités : la **procédure** `Type'Output` et la **fonction** `Type'Input` réservées pour les tableaux ou les types à discriminants permettant aussi de stocker les bornes des tableaux ou le discriminant. Il existe aussi un paquetage `Ada.Text_Io.Text_Streams` pour des fichiers de flots `ASCII`. Voyons l'application (TP16) `T_Appartement`. Tout cela avec des paquetages évidemment.
Rappel : nous aurons défini une structure de base `T_PIECE` (type étiqueté ⇒ racine de la classe), nous aurons dérivé cette structure pour `T_PIECE_D_EAU`, `T_PIECE_SALON`, `T_PIECE_CHAMBRE` et `T_DEPENDANCE`. Chaque structure dérivée possède ses méthodes propres avec entre autres `Get` et `Put` (clavier et écran).

Le type `T_Appartement` est déclaré (simplifié ici) par composition ainsi :

```
type T_Appartement is new Limited_Controlled with record
  CUISINE   : T_Piece_D_Eau ;
  BAINS     : T_Piece_D_Eau ;
  SALON     : T_Piece_Salon ;
  CHAMB     : T_Piece_Chambre ;
  CAVE      : T_Dependance ;
  NUMERO    : Positive ;
…..
end record ;
```

on remarquera en plus que le type est dérivé du type `Limited_Controlled` pour hériter des méthodes « constructeur et destructeur » (`Initialize` et `Finalize`).
Nous définirons des entrées sorties (fichiers de flot) ainsi :

```
procedure ECRIRE_APPART (PTR_FICH : access Root_Stream_Type'Class ;
                         L_APPART : in T_Appartement) ;
for T_Appartement'Write use ECRIRE_APPART ;

procedure LIRE_APPART (PTR_FICH : access Root_Stream_Type'Class ;
                       L_APPART : out T_Appartement) ;
for T_Appartement'Read use LIRE_APPART ;
```

dans le corps du paquetage une réalisation pourra être celle ci :

```
procedure ECRIRE_APPART (PTR_FICH : access Root_Stream_Type'Class ;
                         L_APPART : in T_Appartement) is
begin
   T_Piece_D_Eau'Output(PTR_FICH, L_APPART.CUISINE) ;
   T_Piece_D_Eau'Output(PTR_FICH, L_APPART.BAINS) ;
   T_Piece_Salon'Output(PTR_FICH, L_APPART.SALON) ;
   T_Piece_Chambre'Output(PTR_FICH, L_APPART.CHAMB) ;
   T_Piece_Dependance'Output(PTR_FICH, L_APPART.CAVE) ;
   POSITIVE'Write(PTR_FICH, L_APPART.NUMERO) ;
 …..
end ECRIRE_APPART ;
```

Notez bien les `'Output` et le `POSITIVE'Write`

La procédure `LIRE_APPART` a l'allure :

```
procedure LIRE_APPART (PTR_FICH : access Root_Stream_Type'Class ;
                       L_APPART : out T_Appartement) is
begin
   L_APPART.CUISINE := T_Piece_D_Eau'Input(PTR_FICH);
   L_APPART.BAINS := T_Piece_D_Eau'Input(PTR_FICH) ;
   L_APPART.SALON := T_Piece_Salon'Input(PTR_FICH) ;
   L_APPART.CHAMB := T_Piece_Chambre'Input(PTR_FICH) ;
   L_APPART.CAVE := T_Piece_Dependance'Input(PTR_FICH);
   POSITIVE'Read(PTR_FICH, L_APPART.NUMERO) ;
 …..
end LIRE_APPART ;
```

Notez bien les `'Input` et les affectations (car `'Input` est une fonction) et le `POSITIVE'Read` (qui est une procédure)

A finir et à tester ! TD16.

TD arithmétique (numérique et mantisse)

Bien que placé dans le polycopié cours Ada **ce document sert en TD** (fait en 4 heures semaine 8)

Avertissements :

Le terme « arithmétique » est totalement consacré aux **numériques** (entiers (signés et modulaires) et réels (flottants, fixes et décimaux)). Pour comprendre la modélisation des ces objets numériques il est indispensable de connaître un peu de choses sur **leur implémentation possible** (même si on essaie, grâce à Ada, de **raisonner en faisant abstraction des réalisations** en machine). L'objectif de ce TD (4 heures) est d'apprendre (ou de revoir) la numération binaire (et de réfléchir sur les conséquences d'une réalisation plutôt qu'une autre). Les deux cours (numériques I et II) qui suivront (semaines 9 et 10) **auront besoin des concepts et des notions** vus dans ce TD ! Prenez des notes sous la conduite de votre enseignant. Il est utile de posséder les listings des paquetages `Standard` et/ou `Ada.Characters.Latin_1` pour réaliser certains exercices et pour plus d'efficacité (voir le polycopié « paquetages … »).

Concepts :

Ecriture en base 2 d'un numérique entier (non signé puis signé) puis écriture d'un réel (flottant), notion d'octet (byte), de bit, écriture hexadécimale, écriture normalisée d'un nombre en base quelconque (mais surtout en base 2), mantisse, exposant, caractéristique. **Ce TD n'est pas de l'Ada à proprement parlé**. Il servira ensuite cependant dans les cours Ada (comme signalé dans l'avertissement).

Représentation d'une valeur « non signée » en base 2 :

Exemple : la valeur (ou rang) d'un caractère `LATIN_1`. Cette valeur (voir dans le paquetage `Standard`) va de 0 à 255 pour les caractères (à l'exclusion des `Wide_Character`). La notion de base 2 a déjà été un peu vue dans le cours Ada n°1 (littéraux numériques) et aussi dans la multiplication égyptienne (algorithmique). La décomposition d'une valeur décimale « entière » en base 2 se fait avec les restes successifs de la division par 2 de la valeur. Prenons le caractère `LATIN_1.SPACE` ou encore le caractère blanc, sa valeur (ou rang) est 32 (voir paquetages). En divisant successivement par 2 on obtient la suite des restes 0,0,0,0,1 la valeur s'écrit donc $10000_{(2)}$. Ou encore $1 * 2^4 + 0 * 2^3 + 0 * 2^2 + 0 * 2^1 + 0 * 2^0$!

Le chiffre 1 ici est appelé le bit de fort poids

A faire en TD

Exercices (à faire scrupuleusement en TD) :

1. Quel est le caractère `LATIN_1` de valeur égale à 111000 ? et à 01011011 ? Voir quelques accentués ?
2. Quelle est la valeur `LATIN_1` du caractère '0' ? En décimal, puis en binaire. Les valeurs allant de 0 à 255 peuvent se « coder » sur 8 bits (**bit** ≡ **bi**nary digi**t**). Cet ensemble de 8 bits est appelé **octet** (ou byte) en effet vérifiez que 255 = 11111111 : on arrive à la saturation d'un octet. En remarquant que 1111 qui vaut 15 en décimal ou F en hexadécimal (4 bits saturés) on voit que l'on peut « coder » une valeur d'octet à l'aide de deux chiffres hexadécimaux. Vérifiez en lisant dans une table par exemple à la fin de ce texte. **Refaites les exercices précédents en hexadécimal**.
3. Ecrire simplement (à l'aide de 2^α) le nombre 111111....11111 en base 2 (n chiffres binaires tous à 1). Valeur de α en fonction de n ? Commencez par 11 ($2^2 - 1$) puis généralisez.
   Retenez cette valeur $2^n - 1$ pour le cours numériques 1 !

Représentation d'une valeur « signée » en base 2 :

Dans ce cas le signe (positif ou négatif, donc deux états distincts), est symbolisé par la valeur binaire 0 ou 1 (respectivement). Ce chiffre binaire est placé (convention !) en « fort » poids de la valeur codée. Ainsi :

010100 vaut +20.

le premier bit de gauche est appelé le bit de signe (ici 0), ce n'est plus le bit de « forte valeur » comme précédemment

Combien vaut en représentation binaire –20 ? Une solution (possible !) 110100 n'est pas utilisée (ou peu utilisée) et on lui préfère la représentation **en complément à 2**. La valeur de -20 est écrite de telle sorte que l'addition avec +20 donne zéro (logique !).

symétrique codage binaire de 20 à trouver !

A faire en TD

Donc :

010100 + xxxxxx = 000000

De proche en proche à partir de la droite on trouve xxxxx0 puis xxxx00 puis xxx100 puis xx1100 puis x01100 enfin 101100. Attention au report car 1 + 1 = 10 !

Algorithme de construction d'une valeur symétrique (ou opposée) d'une valeur binaire donnée :

En parcourant, de droite à gauche, la valeur dont on cherche le symétrique, laisser invariants les bits jusqu'à et y compris le premier bit 1 rencontré puis ensuite changer systématiquement les 0 en 1 et les 1 en 0.

Exercice :

Coder le symétrique (ou opposé) de 00001. (Rép : 11111). C'est -1 mais ici sur 5 bits ! Et dans un octet ?

Remarque : 11111111 vaut donc -1 dans un octet « signé » mais vaut 255 dans un octet « non signé » ! Dur !

Ecrire 77_{10} en base 2 (mais dans un octet donc sur 8 bits) puis -77_{10} (toujours dans un octet). L'octet est alors considéré comme signé. Notez bien qu'il ne s'agit plus de codage `LATIN_1`, en Ada la déclaration (donc avec typage) permet de différencier cela.

Même exercice (77 et -77) sur un « mot » de 16 bits. D'autres exemples ?

Quelle est la valeur maximale d'un entier signé **positif** sur 16 bits (en général des `Short_Integer`). Puis, même question, combien vaut 0111111111111111. Utilisez un exercice précédent (**2^n - 1**).
Partez de (et retenez) 2^{10} = 1024 appelé le kilo-informatique (K). Puis 2^{11} = 2048, ...= 4096, ...= 8192, ... =16384, 2^{15} = 32768. (Donc la réponse est : 32767). Codez alors –32767 ? (Rép : 1000000000000001).

Quelle devrait être la valeur (négative) de 1000000000000000 ? Quel est son symétrique ? Lui même ! Quel est le symétrique de zéro (appliquez l'algorithme !) c'est évidemment lui même (logique !). On a donc deux représentation pour coder zéro ! C'est une de trop ! Il est souvent convenu de coder le plus grand entier négatif avec la représentation 1000000000000000. Sur 16 bits c'est donc -32768. Il y a alors dissymétrie entre les positifs et les négatifs (qui ont un élément de plus !).

Ecriture d'une valeur fractionnaire « non signée » en base 2 :

Remarque : on ne s'intéresse plus au problème de la représentation mémoire mais à **des écritures**.
exemple : 0,5 en base 10 (noté 0.5 en Ada !) s'écrit 0.1 en base 2. Pourquoi ?
En effet 0.5 = ½ (évident !) donc 2^{-1} (re-évident !) donc 0.1 en base binaire (notez le point décimal).

© D. Feneuille, TD (Numérique et Mantisse) 2002 (fichier TD_MANTI.DOC) 12/07/02

De même (montrez) : $0.75_{(10)} = 0.11_{(2)}$ C'est le passage de l'écriture base 10 à l'écriture en base 2.

Généralisation. Exemple puis disposition pratique :

Soit 0.6875 (en base 10) à écrire en base 2.
Posons 0.6875 = N alors on a 2*N = 1.375 (évident !) ou 2*N = 1 + 0.375 ou N = (1 + 0.375)/2 donc :

N = ½ + ½ *(0.375) = $0.1_{(2)}$ + ½ *(0.375) puis en recommençant avec 0.375 = ½ * (0.75) et en reportant
N = $0.1_{(2)}$ + ¼ *(0.75) = $0.1_{(2)}$ + 1/8 *(1.5) = $0.1_{(2)}$ + $0.001_{(2)}$ + 1/8 *(0.5) = $0.1011_{(2)}$

Vous verrez avec votre enseignant une **disposition pratique** très connue à **utiliser** systématiquement.

Exercices :

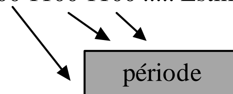
- Coder 12,6875 en base 2. (réponse = 1100.1011)
- Montrez que 0,009765625 = $0.000000101_{(2)}$

Remarquez parfois le peu de chiffres binaires pour « beaucoup » de chiffres décimaux. L'inverse est aussi vrai, il **n'y a pas de règle** ! On verra aussi le premier exercice suivant.

- Représentez 0,9. Problèmes ?

On en déduira la «difficulté» de représenter (donc de coder) un nombre à virgule en mémoire. La perte d'information peut être estimée en fonction du nombre de bits de stockage.

- Soit la représentation « infinie » suivante 0.1 1100 1100 1100 ..... Estimez la perte d'information quand on arrête la « suite » à n chiffres après la virgule.



- Représenter le nombre $0.111111.....11111_{(2)}$ (avec n chiffres après la virgule et tous à 1) utilisez une expression simple du type 2^{α}. Commencez par 0.11 (comme en page 1). Retenez cette écriture **$1 - 2^{-n}$** (utilisée aussi en cours !)

Représentation possible d'un réel en machine (mantisse et exposant):

Pour comprendre les problèmes de l'implémentation des nombres réels (en Ada ou dans tout autre langage) il faut assimiler les notions de **mantisse, d'exposant et de caractéristique.** Aussi, allons nous nous arrêter un instant sur ces concepts. (Ce n'est donc toujours pas, a priori, de l'Ada).

Soit le littéral décimal négatif -3.5 :

- 3.5 en base 10 peut s'écrire en base 2 : -11.1 (rappel)
remarque (à méditer mais hors sujet !) : de même on peut écrire $-3.5_{10} = -3.8_{(16)}$ car 1/2 = 8/16.

Ce nombre (-3.5) doit être stocké en mémoire. **Donc sous forme binaire**. De multiples représentations ont été proposées par des comités de normalisation. Nous allons examiner **à titre d'exercice** la norme IEEE P754 qui date de 1983 et qui a été très utilisée dans de nombreuses implémentations (notamment en PASCAL) et reste une bonne base de réflexion pour les problèmes de représentation des nombres en machine. Ce n'est pas de l'Ada (bis !). On retiendra de ces exercices **seulement l'intérêt pédagogique**.

© D. Feneuille, TD (Numérique et Mantisse) 2002 (fichier TD_MANTI.DOC) 12/07/02

Tout d'abord un nombre réel décimal est transformé en base 2 (sous la forme dite normalisée) ainsi :
$\pm$ 0.1xxx....xxxx * 2e.
Il est important que la partie après 0. que l'on appellera la mantisse **commence toujours par 1** (sauf pour le nombre nul) ceci conditionne alors la valeur de e (l'exposant correctif). Voyons notre exemple:

- $3.5_{(10)}$ = $-11.1_{(2)}$ est transformé en -0.111 * 2^2 dans la forme dite normalisée $\pm$ 0.1xxx....xxxx * 2e.

d'où : mantisse = 111 et l'exposant = 2 (ne pas confondre ici l'exposant 2 avec la base 2 !)

La valeur e, qui s'appelle l'exposant, quand elle est, elle même, codée en machine (donc en binaire) s'appelle alors la caractéristique.

De même : $0.009765625_{(10)}$ qui s'écrit (voir les calculs précédents!) : $0.000000101_{(2)}$ sera sous forme normalisée représenté ainsi : 0.101 * 2^{-6} avec mantisse = 101 et exposant = -6. L'exposant -6 sera à coder en binaire pour devenir la caractéristique (par exemple 1010 sur 4 bits).

Problème des chiffres significatifs :

La notion de chiffres significatifs **est à tort** assimilée, par les étudiants, à la grandeur d'une valeur ou encore plus grave au nombre de chiffres après la virgule. **Il n'en est rien**. Prenons deux exemples :

4056780000 $\Rightarrow$ les chiffres significatifs sont ici 405678
0,000005604 $\Rightarrow$ tandis qu'ici ce sont 5604

On voit que l'on retrouve l'esprit de la mantisse (normalisée) pour illustrer le concept de chiffres significatifs.

Une valeur est donc pleinement représentée par ses chiffres significatifs pondérée par un facteur exposant :
4056780000 = 405678 * 10^4
0,000005604 = 5604 * 10^{-5}

Voyons maintenant le codage en machine donc le passage en forme binaire. Si l'on désire stocker des nombres décimaux à virgule de k chiffres significatifs combien faut-il **au moins** de bits t en mantisse ?

Travaillons sur les "**mantisses**" maximales.
On a d'un côté : 10^k -1 = le plus grand nombre possible en base 10 (exemple k = 3 $\Rightarrow$ 999 = 10^3 -1).
De l'autre côté si t est le nombre de bits pour coder la mantisse équivalente alors, 2^t -1 est la valeur de la plus grande mantisse et on a la relation à respecter :

$$10^k - 1 \leq 2^t - 1 \qquad \Rightarrow \qquad 10^k \leq 2^t$$

et avec la formule connue : $u^v = e^{v*\mathbf{ln}(u)}$ (où **ln** = logarithme népérien) on a : $e^{k\mathbf{ln}10} \leq e^{t\mathbf{ln}2}$ soit :

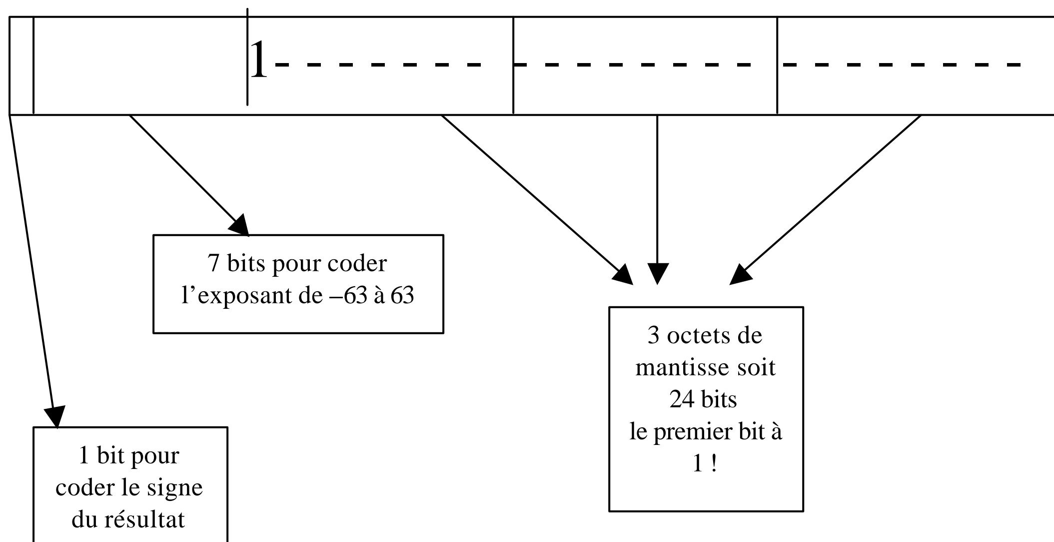
$$k\mathbf{ln}10 \leq t\mathbf{ln}2 \quad \text{d'où :} \qquad t \geq k*(\mathbf{ln}10/\mathbf{ln}2) = k*3,3219...$$

En résumé **t ³ k*3,3219...** (il faut un peu plus de 3 bits pour coder un chiffre décimal significatif)

Exemple : k = 7 $\Rightarrow$ 3,32k = 23,24 $\Rightarrow$ t = 24 bits au moins (3 octets).

Remarque (historique !) :
Ce "k=7", c'est-à-dire 7 chiffres décimaux significatifs, a longtemps été la "**précision**" des petits réels qui utilisaient 4 octets dans certaines implémentations de langages micro (le 4ième octet étant utilisé pour le signe du nombre (1 bit) et les 7 autres bits pour le codage binaire de l'exposant (soit 6 bits de valeur absolue permettant de coder l'exposant de –63 à +63 ! A vérifier !). Voyons un schéma :

Schéma d'un codage sur 4 octets (pour fixer les idées) :



Remarques (la deuxième sera bien commentée en TD) :

1. Dans la norme évoquée plus haut (IEEE) on profite **parfois** (dans certaines implémentations) du fait que le premier bit de la mantisse soit implicitement toujours 1 pour coder, à cet endroit, le signe du nombre, d'où un gain de un bit pour l'exposant qui passe à 8 bits (de -127 à +127). Ceci n'est pas toujours réalisé ! Nous n'en parlerons pas dans les exemples et les exercices.

2. La caractéristique n'est pas forcément codée en complément à deux (comme on l'a vu pour coder un entier signé). L'exposant le plus petit (-63) est alors, par convention, codé 0000000 (exposant sur 7 bits); l'exposant 0 sera codé 63 (0111111); et l'exposant +63 est codé 126 (1111110); ce 63 est appelé le **biais**. Le biais est donc la valeur du codage de l'exposant nul et c'est aussi la valeur que l'on soustrait au codage pour avoir la vraie valeur de l'exposant. (la valeur restante c'est à dire 1111111 soit 127 est réservée pour matérialiser des valeurs particulières ou aberrantes).

Un exemple encore :

k = 11 (11 chiffres significatifs réalisation "possible" des réels micro de certains langages anciens) implique t ≥ 37 bits (à vérifier avec la formule). En général 6 octets (48 bits) sont utilisés pour l'ensemble du codage complet et la caractéristique peut occuper alors un peu plus de 8 bits.

Exercice (hors remarque 1 ci dessus) :

Soit 12,6875 (à mettre sous forme normalisée) donnez le codage résultat en hexadécimal et dans une configuration IEEE à 4 octets. (cf. schéma plus haut).

Aides :

A développer en TD

$12{,}6875 = 1100{,}1011_{(2)}$
$12{,}6875 = 0{,}11001011 * 2^4$ (normalisé)

mantisse ≡ 11001011000...0000 avec 24 chiffres binaires (3 octets).
caractéristique ≡ 4 + biais ≡ 100 + 0111111 ≡ 1000011 avec 7 chiffres binaires
0 / 1000011 / 1100101100..0000 ≡ signe / caractéristique / mantisse
Soit : 43CB0000 en hexadécimal (utilisez le tableau en fin de document page 10).

Exercice : même question avec –0,009765625

-0,009765625 ≡ -0.000000101$_{(2)}$ = -0.101*2^{-6}
mantisse ≡ 10100....000
caractéristique ≡ -6 + biais ≡ 0111111 - 110 ≡ 0111001 (faire la soustraction)
1 / 0111001 / 101000...00 (sur 4 octets)
B9A00000 en hexadécimal

Exercice :

Dans la norme IEEE sur 4 octets, présentez le plus petit réel positif non nul, puis le réel immédiatement supérieur à 1.0, puis le réel positif le plus grand. Donnez les écritures en hexadécimal. Calculez leur valeur en décimal.

Aides :

- plus petit positif :

mantisse 10......0 plus petite mantisse
caractéristique 00......0 plus petite caractéristique
plus petit réel positif : 0 / 0000000 / 1000...000 ⇒ 00800000 hexadécimal
valeur 2^{-64}

- suivant de 1.0 :

mantisse 100...001
caractéristique 100...000
0 / 1000000 / 1000.....0001 ⇒ 40800001 en hexadécimal
valeur : $1 + 2^{-23}$

- plus grand :

mantisse 111....111
caractéristique 111....1110
0 / 1111110 / 111....111 ⇒ 7EFFFFFF en hexadécimal
Valeur : $2^{63} - 2^{39}$

A développer en TD

Remarque : cette valeur de réel Max est entière !

Exercice :

Montrez que l'on a toujours la relation : 0,5 ≤ mantisse < 1

Exercice :

Coder le **réel** 1.0 dans une implémentation à 4 octets (vue précédemment).
Coder **l'entier** 1 (sur 4 octets mais ce n'est plus la même chose).
Comparez et imaginez l'affectation `Le_Reel := FLOAT(1);`

Aides :

1.0 ≡ mantisse : 10 ...... 0 ≡ 0.1 base 2 ≡ 0.1 * 2^1 d'où exposant = 1 ⇒
caractéristique : 10..... 0 ⇒ codage 0 / 100...000 / 1000.....0000

1 ≡ 00000…..00000001 (4 octets). La conversion est plus qu'un simple transfert d'octets !

Valeur de l'intervalle « trou » entre 2 réels contigus :

Soit le réel normalisé 0.100------0 * $2^{\&}$ (avec t bits de mantisse).
Le réel immédiatement supérieur s'écrit : 0.10------01 * $2^{\&}$. Notez le 1 supplémentaire en « bout » de mantisse.

L'intervalle en **valeur absolue** est égale à :

$$0.000\text{------}1 * 2^{\&} = 0.1 * 2^{\&-t+1} = 2^{\&-t}$$

en notant : t le nombre de bits de la mantisse et & la valeur de l'exposant

Exemple : pour fixer les idées avec t = 24 bits (mantisse de 3 octets) et $-63 \leq \& \leq +63$:

On voit que le **trou « absolu » vers les plus grands nombres** est $2^{63-24} = 2^{39}$
Le trou est grand environ 10^{10} mais les nombres sont très grands !

Pour le **trou « absolu » vers les très petits nombres** (proches de zéro) on a &= -63

soit $2^{-63-24} = 2^{-87}$ soit un trou ridicule ! Ce nombre lui même n'est d'ailleurs pas représentable en machine on parle "**d'underflow**".

Exercices (hors norme IEEE) :

Exercice 1 :

Quel est le plus petit nombre en valeur absolue (et non nul) représentable dans cette implémentation

Réponse : MIN = 0.1 * $2^{-63} = 2^{-64}$ (à comparer avec 2^{-87}) étonnant ? non ?

immédiatement en dessous il y a zéro d'où un trou de 2^{-64} entre zéro et ce MIN non nul.

Immédiatement au dessus il y a : 0.10-------01 * 2^{-63} avec le trou de 2^{-87} comme on vient de le voir.

Etonnant oui ! ! !

A retenir : Le **trou absolu** est donc parfois grand parfois très petit. Il dépend de la grandeur des nombres.

Mais qu'en est il du **trou relatif** (ne pas confondre avec le trou absolu !) entre deux réels R1 et R2 ?

Ecart ou trou « relatif » entre deux réels contigus :

La définition du trou **relatif** est :

abs((R1 - R2)/R1) ou **abs**((R1-R2)/R2) c'est-à-dire **abs**(trou absolu/l'un des 2 nombres)

a) voyons le cas où R1 et R2 sont proches de 0.1* $2^{\&}$ (mantisse **petite** et ceci quel que soit l'exposant &)

Le trou absolu est : $2^{\&-t}$ (on l'a vu) et R1 ou R2 vaut ≈ 0.1*$2^{\&}$ donc :

Le trou relatif vaut : $2^{\&-t} / 0.1*2^{\&} = \mathbf{2^{-t+1}}$

b) voyons maintenant vers 0.111....11111111*$2^{\&}$ (mantisse **saturée** et toujours quel que soit &)

Le trou relatif vaut : $2^{\&-t} / 0.111...11111111*2^{\&} \approx \mathbf{2^{-t}}$ car 0.111111.....111111 ≈ 1

On remarque donc que le **trou relatif ne fait pas intervenir l'exposant** ce qui était évident ! On remarque aussi que le trou relatif oscille entre 2^{-t} et 2^{-t+1} ce qui fait **très peu de différence**. On verra le détail des calculs en TD mais on retient que le **trou relatif peut être considéré comme quasi constant partout**.

Exercice 2 :

Soit des réels supposés être représentés avec 5 bits de mantisse seulement et un exposant compris entre -20 et +20. Pour bizarre que puisse être cette représentation on verra en cours qu'il s'agit des réels Ada (**digits** 1) en nombre contrat. En clair ce serait des réels à 1 chiffre significatif seulement ! Folklorique mais !

a) Quel est le plus petit entier positif non représentable dans cette convention ? Essayez d'écrire 1, 2, 3, 4, etc. il ne faut pas aller trop loin pour trouver.

b) Quel est le plus grand réel positif dans cette convention. Est-il entier ?

Aides :

a) 1 est codé $0.1 * 2^1$ ==> mantisse = 10000 ; exposant = 1

2 est code $0.1 * 2^2$ ==> mantisse = 10000 ; exposant = 2

7 est codé $0.111 * 2^3$ ==> mantisse = 11100 ; exposant = 3

15 est codé $0.1111 * 2^4$ ==> mantisse = 11110 ; exposant = 4

31 est codé $0.11111 * 2^5$ ==> mantisse = 11111 ; exposant = 5

32 est codé $0.10000 * 2^6$ ==> mantisse = 10000 ; exposant = 6

33 est codé $0.100001 * 2^6$ ==> mantisse = impossible trop de bits

la réponse est 33

b) $0.11111 * 2^{20} = 11111 * 2^{15}$

$(2^5-1) * 2^{15} = 2^{20} - 2^{15}$ ce grand réel est donc un entier !

Conclusion :

Il est clair qu'avec de tels problèmes les calculs, d'un langage à l'autre et d'une machine à l'autre, ne soient pas identiques pour un même algorithme donné ⮚ pas de portabilité et ceci est inadmissible. Vivement Ada pour améliorer un peu cela !

A lire (hors TD) pour se préparer au cours qui suivra :

D'abord il y a la théorie et les réels mathématiques qui "veut" qu'entre deux réels quelconques il en existe toujours une infinité ! Impossible à représenter en machine avec un nombre fini de bits (donc un nombre fini de valeurs). On ne pouvait déjà pas représenter l'infinité des entiers. Alors !

Mais les nombres "réels-ordinateur" ne servent qu'à faire des calculs avec des données mesurées au sens physique du terme c'est à dire soumises à une incertitude (ou définies avec une certaine précision). En machine aussi on parle de précision. Cette précision sur la donnée en machine peut prendre deux aspects :

absolue : on connaît une borne supérieure de la valeur absolue de l'incertitude.

Exemple : les calculs comptables (au centime près pour tous les cumuls).

relative : on connaît une borne supérieure pour la valeur absolue du quotient de l'incertitude par la grandeur (on parle alors de nombre de chiffres significativement exacts).

Ada et les réels (premier survol vu en détail en cours) :

Toutes les implémentations d'Ada proposent 3 types pour représenter les nombres à virgule :

- le type dit "**digits**" (réels point-flottant) où l'on garantira le nombre de chiffres significatifs mais le "**trou**" entre 2 nombres "contigus" dépend de la valeur de ces 2 nombres (on parle d'incertitude relative). FLOAT (type prédéfini) est de cette famille (**range** est facultatif).
- le type dit "**delta**" (réels point-fixe) où le "**trou**" entre 2 nombres est constant. **La plage de valeur (range) de ce type est obligatoirement un intervalle** (DURATION est de cette famille).
- le type dit "**delta**" "**digits**" (réels décimaux) où le "**trou**" entre 2 nombres est lui aussi constant. **La plage de valeur (range) de ce type est facultative**. Réservés aux calculs comptables.

Ou encore dit autrement :

- **type réel point-flottant :** correspondant à un nombre de chiffres total fixé donc correspondant à une **précision relative** (réels dits "**digits**").
- **type réel point-fixe :** correspondant à un nombre de chiffres après la virgule fixé donc correspondant à une précision absolue (réels dits "**delta**")
- **type réel décimaux :** correspondant à un nombre de chiffres après la virgule fixé donc correspondant à une précision absolue avec, en plus, un nombre de chiffres décimaux total fixé (réels dits "**delta**" "**digits**").

Remarque :

Il existe, comme avec le type entier, un **type réel universel** possédant des propriétés similaires c'est-à-dire compatible avec tout (ou presque).

Exemple :

PI : **constant** : = 3.14159_26536 ; on ne précise pas le type ⇒ donc c’est un réel universel

Les littéraux avec point décimal sont des réels universels. La constante PI ci-dessus est cependant prédéfinie dans le paquetage Ada.Numerics avec beaucoup de chiffres significatifs.

Un **réel** universel peut être multiplié par un **entier** universel et inversement.
Un **réel** universel peut être divisé par un **entier** universel et réciproquement.
Le résultat sera **toujours** du type **réel universel**.

© D. Feneuille, TD (Numérique et Mantisse) 2002 (fichier TD_MANTI.DOC) 12/07/02

Exemples :

```
DEUX_PI :        constant := 2* PI; -- ok
DEUX_PI :        constant := 2.0*PI; -- ok
PI_PLUS_DEUX :   constant := PI+2; -- illégal
```

Notez !

```
DEUX : INTEGER := 2;                  -- INTEGER
E : constant : = 2.71828_18258;       -- réel universel
MAX : constant := 100;                -- entier universel

a) DEUX * E           -- illégal
b) DEUX * MAX         -- INTEGER
c) E * MAX            -- Réel universel
d) DEUX * DEUX        -- INTEGER
e) E * E :            -- Réel universel
f) MAX * MAX          -- entier universel.
```

Modélisation des réels point-flottant (et en résumé) :

La représentation des nombres flottants se heurte à la difficulté qu'il y a à couper les bits en fractions. On structure le réel point flottant en cinq parties (ou champs) :

- Signe mantisse (ou signe du nombre)
- Mantisse elle même
- Exposé (ou base) souvent égal à 2 (voire 16)
- Signe de l'exposant
- l'exposant lui-même

La mantisse est implicitement comprise dans l'intervalle [0.5..1.0[d'où son codage binaire entre 0.1 et 0.111111.....1111111. En machine le 0 disparaît et le point aussi! Il faut coder et gérer ces 5 champs. La normalisation empêche de représenter 0.0 et conduit à faire un codage spécial.

La variété des représentations possibles rend le portage aléatoire (pour des applications numériques). Ainsi (même en Ada) il sera **moins aisé de parler de portabilité totale** avec les réels comme cela avait été le cas avec les entiers.

Conversions

décimal	binaire	hexadécimal
0	0000	0
1	0001	1
2	0010	2
3	0011	3
4	0100	4
5	0101	5
6	0110	6
7	0111	7

décimal	binaire	hexadécimal
8	1000	8
9	1001	9
10	1010	A
11	1011	B
12	1100	C
13	1101	D
14	1110	E
15	1111	F

© D. Feneuille, TD (Numérique et Mantisse) 2002 (fichier TD_MANTI.DOC) 12/07/02

Cours n°1 les numériques (2 heures)

Présentation des concepts :

Enseigné en 20 heures réparties de la façon suivante (voir découpage plus loin) :

- 4 heures de TD **déjà effectuées avant** ce premier cours (cf. document page précédentes)
- 4 heures de cours (faites en 2 séances de 2 heures chacune).
- 12 heures de TP (4h probabilité, 4h intégrale, 4h tests boîte claire).

Les concepts enseignés portent sur :

- Les types dérivés et les types formellement définis comme dérivés. Retour sur l'héritage (hors celui des classes). Application de l'héritage aux numériques.
- Surcharge d'opérateurs et T.A.D. (encore !). Clauses de représentation (`for` …`use`…).
- La portabilité **totale** des entiers et la portabilité **partielle** des réels.
- Les types numériques réels flottants et fixes (problèmes de codage mémoire : mantisse, exposant).
- La solution Ada avec le type "`digits`", le type "`delta`". "et le type `delta-digits`" (et leurs attributs).
- La différence entre convergence mathématique et convergence informatique.
- Précision relative et précision absolue, incertitude, test d'arrêt dans les algorithmes numériques.
- Problèmes de troncature, perte d'information et divergence des algorithmes codés.

Le langage associé est bien sûr Ada !

<u>Découpage : (2002)</u> (voir grille générale)

<u>Semaine 8</u>: TD (numériques et mantisse) <u>4h nécessaires pour ce cours</u>

<u>Semaine 8 ou 9</u> : Cours n°1 (types digits, delta et delta-digits) (2h) celui-ci !

<u>Semaine 11</u>: TP sur les probabilités (4h). Cours n°2 (Convergence et précision) (2h)

<u>Semaine 12</u> : TP 15 sur l'intégration (4h).

<u>Semaine 14</u> : tests boîte claire (4h) ou TD-TP 19 de synthèse.

Types Dérivés (et retour sur le fort typage)

Introduction (reprise de concepts déjà largement évoqués dans les cours antérieurs) :

Il est utile parfois d'introduire un nouveau type, semblable dans ses caractéristiques à un type existant, mais néanmoins distinct de celui-ci (c'est le principe du **fort typage**). Nous avons déjà mis ceci en évidence, dans de nombreux cours et pas seulement au cours n°10 avec les objets. Rappelez-vous, au début, dans les généralités ou dans le cours n°2 avec les types scalaires :

```
type T_JOUR is range 1..31 ;       -- par exemple
```

Avec cette déclaration le type `T_JOUR` est un type entier du même « genre » que le type `Integer` mais plus **restreint** et **surtout incompatible**. Notez bien, une fois de plus, la déclaration **`type`** et non pas **`subtype`** ainsi que **l'absence de référence à un type de base**. Seule la définition de l'intervalle (**`range`**) évoque un type possible (ici entier) mais c'est un choix de la machine et non du concepteur du programme. C'est donc **un type construit** qui est **dérivé** d'un des types entiers **de la machine**. On dit qu'il est **formellement dérivé**.

Types formellement définis comme dérivés :

Le manuel de référence (lecture conseillée) utilise la notion de type formellement dérivé pour illustrer la signification exacte de certaines définitions du langage sous l'allure formelle :

```
type DÉCLARÉ_PAR_L_UTILISATEUR is CONTRAINTE_DE_SOUS_TYPE;
```

Exemple : **`type`** `T_INTERVALLE` **`is range`** `-40..80;`

Ce qui équivaut à :

```
type ANONYME is UN_CERTAIN_TYPE ; -- choisi par le compilateur
subtype DÉCLARÉ_PAR_L_UTILISATEUR is ANONYME range CONTRAINTE_DE_SOUS_TYPE;
```

La contrainte peut contenir une expression à condition qu'elle puisse être **évaluée à la compilation**.

Application au cas des types numériques (notamment les entiers) :

Les entiers (et les réels d'ailleurs) de la machine sont réalisés par l'un des types internes sur lesquels opèrent les instructions de la machine. La **réalisation du type** formellement dérivé est **choisie par le compilateur** (la structure en nombre de bits par exemple), celui-ci s'appuie sur un des types prédéfinis mais qui ne porte pas de nom. **Cette réalisation dépend du compilateur et de la machine**. La contrainte de sous type de l'utilisateur est ensuite ajoutée au type réalisé (voir exemple ci-dessous avec les entiers).

Le type `INTEGER` : (type entier **prédéfini**)

Toutes les implémentations (réalisations physiques) d'Ada possèdent le type prédéfini `INTEGER`. Si nous notons n le nombre de bits utilisés pour coder un entier de type `INTEGER` :

alors : $-2^{n-1} \leq$ l'entier $\leq +2^{n-1} - 1$ ((vu ?) en TD)

Sur les premiers PC on avait souvent n = 16 et la valeur des entiers était alors comprise entre -32768 et +32767. Sur une machine (32 bits) (et avec notre GNAT linux) c'est -21474_83648..21474_83647. Sur la plupart des machines les nombres négatifs sont codés en complément dit à 2. L'intervalle des valeurs de ces types prédéfinis n'est donc pas symétrique exactement par rapport à zéro. On peut trouver aussi (pas obligatoire) d'autres types prédéfinis tels que : `LONG_INTEGER` ou `SHORT_INTEGER` mais, là aussi, le nombre de bits est différent d'une machine (et/ou d'un compilateur) à l'autre !

Problème de choix d'entiers (portabilité) :

Soient 2 machines MACHINE_A et MACHINE_B (deux machines imaginaires inspirées de Barnes)

- Avec MACHINE_A, on dispose de 2 types arbitraires.

`INTEGER`	-32768 .. + 32767 (codés sur 16 bits)
`LONG_INTEGER`	-21474_83648 .. + 21474_83647 (sur 32 bits)

- Avec MACHINE_B, nous disposons de 3 types (toujours arbitraires)

`SHORT_INTEGER :`	-2048 .. + 2047 (sur 12 bits)
`INTEGER`	-83_88608 .. + 83_88607 (sur 24 bits)
`LONG_INTEGER`	-14073_74883_55328 .. + 14073_74883_55327 (sur 48 bits)

Remarque : Dans la plupart des problèmes élémentaires le type `INTEGER` est suffisant sur l'une ou l'autre des machines. Mais nous avons d'ailleurs évité d'utiliser, le type `INTEGER` jusqu'à présent dans nos exercices (on va montrer que c'était une bonne stratégie!).

Si nous devons maintenant manipuler une valeur signée disons jusqu'à 1 million, (±1E6) alors le type `INTEGER` ne convient plus sur MACHINE_A (où il faut utiliser le type `LONG_INTEGER`) et sur MACHINE_B il serait déraisonnable d'utiliser ce type `LONG_INTEGER`. On voit ainsi qu'une déclaration systématique `INTEGER` (pourtant facile) pose problème et **n'est pas portable d'une machine à l'autre**.

On surmonte la difficulté de notre problème soit :

1) en **utilisant un type dérivé** avec **`new`** (opérateur déjà vu mais rappelé)

Pour MACHINE_A : `type T_ENTIER is new LONG_INTEGER;`
Pour MACHINE_B : `type T_ENTIER is new INTEGER;`

mais **il faudrait connaître les implémentations** des machines (pas toujours facile, ni pratique)

2) en **laissant le <u>choix automatiquement au compilateur</u>** en écrivant

```
type T_ENTIER is range -1E6..+1E6; -- 1 million
```

L'implémentation choisit alors implicitement le plus petit type approprié

Tout se passe, dans le deuxième cas, comme si nous avions écrit :

Pour A : `type T_ENTIER is new LONG_INTEGER range -1E6..1E6;`
Pour B : `type T_ENTIER is new INTEGER range -1E6..1E6;`

DONC : c'est le deuxième cas ("choix par le compilateur") qui est le plus élégant et évidemment recommandé. A l'avenir faites toujours disparaître les simplistes déclarations systématiques : `INTEGER`, réfléchissez à la structure de vos entiers pour votre application et déclarez vos propres entiers. Pour les entrées-sorties, pas de problème, instanciez `TEXT_IO.INTEGER_IO` avec le type créé. C'est cette manière de procéder qui permet de séparer complètement le programme des particularités de la machine. Ainsi on réalise une totale portabilité sur les entiers. <u>*Cette pratique est fondamentale*</u>. Toujours et encore les préceptes du génie logiciel !

Type dérivé (rappel ! enfin on espère !) :

On voit maintenant la **dérivation explicite** d'un type et ses conséquences. Ce qui suit bien qu'utilisé avec les numériques n'est pas uniquement propre aux types numériques.

De façon générale : si `T_PERE` est un type quelconque (**`tagged`** ou non) , alors, avec la déclaration :

```
type T_NOUVEAU is new T_PERE;
```

On a dérivé le type T_PERE : T_NOUVEAU est un type dérivé de T_PERE et T_PERE est dit le type père de T_NOUVEAU (notez bien le **new**).

Un type dérivé appartient à la même famille de types que son père (si T_PERE est un type article alors T_NOUVEAU sera un type article). L'ensemble des valeurs d'un type dérivé **est une copie** de l'ensemble des valeurs du type père (sauf s'il y a un **range,** en plus, qui limite l'intervalle des valeurs héritées). **Mais on a quand même des types différents** et les valeurs d'un type ne peuvent pas être affectées à des objets de l'autre type. Cependant la conversion entre les deux types est possible. On rappelle aussi que, pour les objets et les classes (cours 10), on dérive avec, en plus, les mots **with record** (associés à **new**).

Soit :

```
LE_NEW      : T_NOUVEAU;
LE_VIEUX    : T_PERE;
```

alors : `LE_NEW := LE_VIEUX;-- interdit`

mais : `LE_NEW := T_NOUVEAU(LE_VIEUX); -- possible avec conversion`

Rappel des règles d'héritage concernant les types dérivés. (voir exemples plus bas)

Les opérations applicables aux types dérivés sont les suivantes :

- Un type dérivé possède les mêmes attributs (s'ils existent) que le type père.
- L'affectation, l'égalité et l'inégalité sont également applicables **sauf** si le type père est **limited**.
- Un type dérivé héritera de **certains** sous-programmes (S/P) ou opérations primitives applicables au type père avec des règles particulières voyons 2 cas :

1er Cas : **Le type père est un type prédéfini** : les S/P hérités se réduisent aux S/P prédéfinis, et si le type père est lui même un type dérivé alors les S/P hérités seront de nouveau transmis.

2ème Cas : **Le type père est déclaré dans la partie visible d'un paquetage** : tout S/P déclaré dans cette partie visible sera hérité par un type dérivé **si ce dernier** (le fils) **est déclaré après** les S/P soit dans le même paquetage soit dans un autre (paquetage fils ou paquetage s'appuyant sur le premier avec **with**).

Exemples :

1er Cas :

```
type T_ENTIER is new INTEGER;
```
dérivation du type prédéfini INTEGER

T_ENTIER hérite de tous les sous-programmes prédéfinis tels que "+" , "**-**", et "**abs**" ainsi que les attributs FIRST et LAST, mais pas de GET ou PUT par exemple. Ces deux derniers S/P ne sont pas prédéfinis (car définis dans le sous paquetage générique ADA.TEXT_IO.INTEGER_IO (revoir cours E/S)).

Si nous dérivons un autre type de T_ENTIER (**type** T_AUTRE **is new** T_ENTIER;) les sous-programmes hérités (les opérateurs prédéfinis entre autres) seront de nouveau transmis.

2ème Cas :

```
package P_TEMPERATURE is
      type T_TEMPERATURE is range -40..+60;
      procedure AJUSTER (T : in out T_TEMPERATURE);
      ...........
      procedure LIRE (T : out T_TEMPERATURE);
end P_TEMPERATURE;
```

Nous avons donc un type entier T_TEMPERATURE déclaré dans la partie visible du paquetage ainsi que les sous-programmes AJUSTER,..., LIRE

Si nous déclarons :

```
      type T_CELSIUS is new T_TEMPERATURE;
```

soit :

- hors paquetage (notamment dans une entité évoquant ce paquetage avec **with**)
- dans la partie privée du paquetage (si le **private** est utilisé)
- dans le corps du paquetage
- dans la partie visible après LIRE.
- Dans un paquetage fils.

alors T_CELSIUS héritera, à l'un de ces endroits des sous-programmes AJUSTER ,..., LIRE ainsi que des S/P prédéfinis ("+", etc.) comme dans le cas n° 1 car T_TEMPERATURE qui est dérivé implicitement du type entier "machine" héritait de "+". Son fils T_CELSIUS hérite également.

Si nous ne sommes pas satisfaits d'un S/P hérité, **abs** par exemple sur les entiers, nous pouvons le redéfinir :

```
package P_TEMPERATURE is
      type T_TEMPERATURE is range -40..+60;

      ........inchangé
      function "abs"(N : in T_TEMPERATURE) return T_TEMPERATURE;
end P_TEMPERATURE;
```

Donc si nous écrivons

```
      type T_CELSIUS is new T_TEMPERATURE;
```

alors la nouvelle version de **abs** (dédiée à T_TEMPERATURE uniquement et non à un type entier "machine") sera héritée.

Une utilisation des types dérivés

Les types dérivés sont nécessaires : quand nous voulons utiliser les opérations de types existants sur de nouveaux types semblables mais surtout quand nous **voulons éviter le mélange accidentel d'objets de types conceptuellement différents**.

L'utilisation judicieuse des types dérivés explicitement (mais aussi et surtout implicitement) peut faciliter le **développement d'applications en grande sécurité** comme on va le voir.

exemple (simpliste pour le moment et encore emprunté à Barnes) :

```
      type T_ENTIER is range 0..1E5;
      type T_POMMES  is new T_ENTIER;
      type T_ORANGES is new T_ENTIER;

         NB_DE_POMMES  : T_POMMES;
         NB_D_ORANGES  : T_ORANGES;
```

T_POMMES et T_ORANGES dérivent de T_ENTIER donc aussi du type entier "machine" (INTEGER ou LONG_INTEGER suivant les implémentations). Ils héritent donc de l'opérateur "+" par exemple.

Nous pouvons écrire :
```
NB_DE_POMMES  := NB_DE_POMMES + 1;
NB_DE_ORANGES := NB_D_ORANGES + 1;
```

Mais nous ne pouvons pas écrire (à cause du compilateur, et c'est heureux conceptuellement!) :
```
NB_DE_POMMES := NB_D_ORANGES;-- illégal.
```

Par contre nous pouvons écrire (en explicitant donc, mais **c'est de notre volonté!**)
```
NB_DE_POMMES := T_POMMES(NB_D_ORANGES);-- conversion
```

Problème (ou une utilisation de la qualification)

Soit les procédures :
```
procedure VENDRE (N : in T_POMMES);
procedure VENDRE (N : in T_ORANGES); --(il y a surcharge !)
```
Nous pouvons écrire sans ambiguïté :
`VENDRE (NB_DE_POMMES);` ou `VENDRE(NB_D_ORANGES);`
Mais l'expression `VENDRE(6);` est ambiguë (il faut **qualifier**) et par exemple écrire :
`VENDRE (T_POMMES'(6));`

Différence entre qualification et conversion :

Soit `T_LONGUEUR` un type **numérique réel** quelconque. En quoi diffèrent les 2 écritures suivantes:

```
:=  T_LONGUEUR (DIST * 3.14);       --   conversion

T_LONGUEUR '(DIST * 3.14)       --  qualification notez le '
```

Le membre de droite de la **première écriture** utilise une **conversion**. Le type de `DIST` est réel quelconque (et non forcément de type `T_LONGUEUR`) le littéral 3.14 est considéré comme de même type que `DIST` et le résultat de l'opération aussi! Ce résultat est enfin converti dans le type `T_LONGUEUR`. **Il est vraisemblable que le compilateur générera du code pour cette conversion.**

L'expression illustrée par la **deuxième écriture** représente une **qualification** comme on l'a déjà vu à propos des types énumératifs (dans ce cas c'était surtout pour lever une ambiguïté car certaines valeurs de type énumératif pouvaient appartenir à plusieurs types). La qualification exprime ici que l'opérateur de `DIST` * 3.14 (donc l'opérateur `*`) a des opérandes de type `T_LONGUEUR` **donc** que `DIST` et 3.14 **sont de types compatibles avec ce résultat**. En fait **on qualifie l'opérateur *** le compilateur vérifie cette contrainte à la compilation et ne génère pas de code supplémentaire car il n'y a pas de conversion à faire. La qualification est **souvent nécessaire** avec les **agrégats** ou avec un **schéma `case`** pour le **sélecteur**.

Problèmes et difficultés :

Introduction :

```
type T_LONGUEUR is new FLOAT;
type T_SURFACE  is new FLOAT;
```

Avec ces deux dérivations explicites nous ne pouvons plus mélanger les longueurs et les surfaces (ouf !) mais hélas nous héritons de "`*`" ce qui permet de multiplier 2 longueurs pour donner ….. une longueur ! et de multiplier deux surfaces pour donner …. une surface ! **Est-ce bien raisonnable** ? Ne devrions nous pas redéfinir le "`*`" pour ces nouveaux types ? Voir exercices ou exemples qui suivent.

Surcharges d'opérateurs prédéfinis :

Soit les types dérivés

```
type T_DISTANCE      is new FLOAT;
type T_VITESSE       is new FLOAT;
type T_TEMPS         is new FLOAT;
type T_SURFACE       is new FLOAT;
```

Le compilateur avec :

```
D1, D2    : T_DISTANCE;
V         : T_VITESSE;
T         : T_TEMPS;
```

refusera

```
D1 * T       -- illégal
```
(et merci ADA de me l'indiquer)

Mais, on l'a vu, certaines opérations **acceptables par le compilateur doivent être redéfinies** parce que conceptuellement sans fondement. Par exemple le produit entre deux distances devra délivrer un résultat de type `T_SURFACE` et non de type `T_DISTANCE` comme cela est le cas "naturellement" à cause de l'héritage. D'autre part des opérations impossibles (a priori) entre des types disjoints doivent pouvoir être proposées à l'utilisateur comme par exemple la division entre une distance et un temps qui donnera une vitesse et le produit d'un temps par une vitesse qui rendra une distance. Enfin il faudrait interdire des opérations syntaxiquement correctes mais absurdes comme le produit de deux vitesses ! Vaste programme !

Certaines redéfinitions sont faciles à écrire par exemple :

```
function "/" (D : in T_DISTANCE; T : in T_TEMPS) return T_VITESSE is
begin
     return T_VITESSE(FLOAT(D)/FLOAT(T));
end "/";

function "*" (V : in T_VITESSE; T : in T_TEMPS) return T_DISTANCE is
begin
     return T_DISTANCE(FLOAT(V)*FLOAT(T));
end "*";
```

Il sera possible alors d'écrire :

```
V := D/T;"
D := V*T ;
```

mais attention on n'a pas `T*V` ! Il faudra encore surcharger "*" et le rendre commutatif !
Soit les déclarations supplémentaires suivantes :

```
type  T_VOLUME               is new FLOAT;
type  T_ACCELERATION         is new FLOAT;
type  T_MASSE                is new FLOAT;
type  T_FORCE                is new FLOAT;
```

et les spécifications :

```
function "*" (G, D : T_LONGUEUR) return T_SURFACE ;
function "*" (G : T_LONGUEUR; D : T_SURFACE) return T_VOLUME;
function "*" (G : T_MASSE; D : T_ACCELERATION) return T_FORCE;
function "*" (G : T_ACCELERATION; D : T_TEMPS) return T_VITESSE;
function "*" (G : T_VITESSE; D : T_TEMPS) return T_DISTANCE;
etc.
```

Soit les déclarations :

```
D : T_DISTANCE;
E : T_VITESSE;
T : T_TEMPS;
V : T_VOLUME;
L : T_LONGUEUR;
```

Alors :

```
D := E * T      --  ok
D := T * E;     --  non il faut définir la commutativité!
V := D * D * D  --  non évident mais ceci :
V := L * L * L  --  non plus ! moins évident ! vu au cours 13
V := L *(L * L) --  oui ! logique non ? Merci Ada!
```

Remarque et solution générale au problème :

Pour interdire des opérations possibles (comme par exemple le produit de deux volumes) il faut déclarer tous ces types dans un paquetage avec la mention **`private`**. De ce fait les seules opérations permises sont (rappel, !) l'affectation et l'égalité. On déclare (en spécifiant) ensuite les opérations souhaitables entre les types. Dans la partie privée plus bas on les dérive comme précédemment. L'utilisateur ne peut alors qu'utiliser les opérateurs (méthodes) qui sont proposés dans la partie visible. Encore le concept de T.A.D. !

Généralités sur les types numériques

Rappel (du TD Mantisse ?)

Il existe deux classes de types numériques

- Types entiers signés et modulaires
- Types réels : Types point-flottant.
  Types point-fixe.
  Types décimaux.

Et les prédéfinis (respectivement) :

`INTEGER` (voire `LONG_INTEGER`, `SHORT_INTEGER`) pour les entiers signés.
`FLOAT` (voire `LONG_FLOAT`, `SHORT_FLOAT`) pour les réels point-flottant.
`DURATION` pour les réels point-fixe. Utile pour le temps réel (paquetage `CALENDAR`)
Pas de prédéfinis avec les décimaux.

Sauf pour le dernier nommé (`DURATION`) il est recommandé de **définir, soit même, le type utilisé** et ceci pour des problèmes de plus grande portabilité.

DIGITS et DELTA

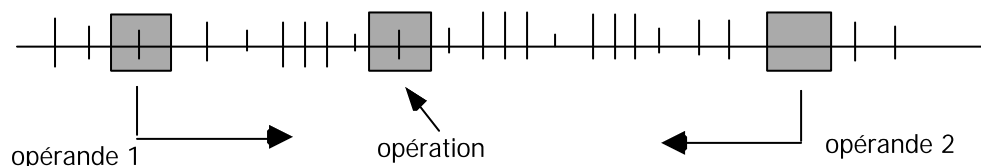
Ce contenu de ce cours postule que le TD de 4 heures (numériques et mantisse) a été enseigné aux étudiants.

Formalisation des réels

Les nombres réels, en informatique donc en Ada, sont presque toujours des nombres **approchés**. Mais certains nombres réels sont représentables de manière exacte (mantisse non périodique par exemple), ils sont alors implémentables (réalisables en machine) si la structure prévue le permet (mantisse suffisante et exposant).

Le programmeur Ada pour travailler passe un contrat avec le compilateur en précisant son type. On verra plus loin comment la déclaration de type "implique" mantisse et exposant minima. Le compilateur va tenter de satisfaire ce contrat en réalisant au moins ce que demande le programmeur. On parle alors de nombre modèle pour cette réalisation (nombre anciennement appelés nombre sûr). La machine fournira une implémentation au moins aussi précise que demandée (ou refusera la contrat).

Mais tout nombre de la réalité ne correspond pas forcément à un nombre modèle (ou nombre réalisé). Le résultat est approché, compris entre deux nombres modèles. La valeur d'un résultat dépend du type et de l'intervalle proposé par la machine. Les opérations agissent sur des nombres pouvant se trouver dans un intervalle, et leur résultat est à son tour vraisemblablement dans un intervalle.



C'est pourquoi **on ne peut parler de portabilité totale,** puisque, pour un même contrat, des machines proposeront des résultats plus poussés que d'autres. Normalement l'effet d'un programme ne pourrait être valablement défini **indépendamment de la machine** qu'en termes "d'intervalles de contrat proposé". Mais en fait, chaque machine peut souvent manipuler pour la précision indiquée par le type un ensemble de nombres plus important que les seuls nombres du contrat, mais cet ensemble dépend de la machine utilisée. Des attributs permettront de connaître les caractéristiques des nombres réalisés pour une machine particulière.

Les nombres réels "point-flottant" :

Toutes les implémentations Ada possèdent un type prédéfini `FLOAT` dont **la réalisation dépend de la machine (au moins une quinzaine de chiffres significatifs)** et peuvent avoir d'autres types prédéfinis :

`SHORT_FLOAT`, `LONG_FLOAT` (voire `LONG_LONG_FLOAT...`) avec plus ou moins de précision.

Ces types possèdent toutes les opérations prédéfinies :

`+, -, *, /,` **`abs`**`,  **` ainsi que :

`=, /=, <, <=, >, >=` attention pas de **`mod`** ni **`rem`** comme pour les entiers !

Sans oublier les attributs `'FIRST`, `'LAST` (intéressants s'il y a une contrainte **`range`**) et l'attribut `'IMAGE` (pour la conversion en `STRING`) et enfin `'PRED` et `'SUCC` (valables sur scalaire donc sur tout numérique).

Nous pouvons dériver tout type prédéfini réel (voir TP arithmétique n°1) avec **new**. Exemple :

```
        type T_REEL is new FLOAT;
ou      type T_PROBA is new FLOAT range 0.0..1.0;
```

Toutefois comme pour les entiers, et pour une **meilleure portabilité**, il est préférable d'indiquer la précision requise et de **permettre à l'implémentation de choisir** de façon appropriée. Par exemple avec :

```
type T_REEL is digits 7; -- 7 chiffres significatifs au moins
```

nous demandons (par contrat) à l'implémentation de dériver `T_REEL` d'un type « machine » ayant **au moins** 7 chiffres de précision.

Un type point flottant construit est donc défini à partir d'une précision relative exprimée par un **nombre de chiffres exacts** (ici 7). Attention (rappel) il ne s'agit pas du nombre de chiffres après la virgule !

Les nombres « contrat » (c'est-à-dire les nombres garantis quelle que soit la mantisse) ont la forme

Signe * mantisse * (base ^{exposant})

La précision demandée (**digits** D) impose un nombre de chiffres minimum pour la mantisse et l'exposant afin de réaliser la précision relative de D chiffres décimaux. Pour la mantisse ce nombre est noté B; ce sont B chiffres après le point dans la représentation dans la Base

Cas général :
soit : `type REEL is digits D;` -- D expression statique

Alors B est au moins égal à l'entier immédiatement supérieur à :

`1 + D * (ln(10)/ln(base))` (formule vue avec IEEE P754)

Si base est fixée à 2 alors : on a $B-1 < 1 + (3.3219 ... * D) < B$

La précision binaire détermine les nombres modèles qui sont par définition d'une part : 0.0 et d'autre part tous les nombres de la forme :

Signe.mantisse.$2^{exposant}$ avec : $0.5 \leq$ mantisse<1
Signe vaut + 1 ou -1
et $-4B \leq$ exposant $\leq +4B$ (c'est la norme Ada)

Chaque machine fournit un nombre de chiffres de mantisse presque toujours supérieur à ce nombre (B). **Entre deux nombres «contrat» ainsi définis, les valeurs supplémentaires correspondent à des nombres implémentés** (qualifiés de **modèles**).

Résumé :

Le compilateur ne crée pas, pour chaque type, une représentation particulière. Il choisit simplement un type plus performant, que celui demandé, parmi ceux qu'il sait manipuler. Si notre compilateur connaît `digits 15`, c'est sur cette forme que sont implémentés les types de précision inférieure comme `digits 4`. Il disposera alors en plus des nombres contrat, de tous les nombres intermédiaires dont il n'a aucune raison de se priver. Ces nombres sont nommés modèles (ou sûrs en Ada 83). Ce sont les nombres représentables sur la machine lorsqu'elle travaille sur un type donné. Enfin pour les entrées-sorties il faut instancier `TEXT_IO.FLOAT_IO` avec le type `digits` déclaré pour utiliser les procédures habituelles `GET`, `PUT` etc.

Quelques exemples de contrats minimum à partir de D : (à vérifier mais calculette oblige !)

(D digits)	(B mantisse)	(bits d'exposant)	(plus petit)	(plus grand)
4	15	7	4.33E-19[1]	1.15E+18[2]
5	18	8	1.05E-22	4.72E+21
9	31	8	2.35E-38	2.12E+37
10	35	9	3.58E-43	1.39E+42
30	101	10	1.21E-122	4.13E+121

Tous les détails des implémentations peuvent être obtenus par le moyen des attributs. En voici quelques uns. Pour tout type ou sous-type F (déclaré **`digits`**), on a (nombres modèles) :

`F'MODEL_MANTISSA`	le nombre de chiffres binaires de la mantisse utilisée
`F'MODEL_EMIN`	l'exposant minimum (négatif ! évident !)
`F'MODEL_SMALL`	le plus petit nombre modèle positif `2.0 * * (F'MODEL_EMIN - 1)`
`F'MODEL_EPSILON`	la différence entre 1.0 et le nombre modèle qui le suit immédiatement : 2.0 * * (1-F'MODEL_MANTISSA)

voir aussi `F'Machine_Emin`, `F'Machine_Emax`, `F'Machine_Mantissa`.

Exemple " hyper simple" (pour mieux comprendre ?) : `type GROSSIER is digits 1;`

supposons que **ce contrat soit réalisé exactement** [3] d'où D = 1 ⇒ B = 5 (1 * 3.3..... + 1)
La précision (relative bien sûr) varie de 1/31 à 1/16 (cf. TD : $2^{-5}/0.11111_{(2)}$ et 2^{-5+1})

Mantisse des nombres modèles :
Les 16 nombres modèles sont des valeurs **dont la mantisse** est l'une des valeurs suivantes (rappel : 0.5 ≤ mantisse < 1) :

16/32	17/32	18/32	......	31/32	(en fraction)
0.5	0.53125	0.5625	......	0.96875	(en décimal)
0.1	0.10001	0.10010	0.11111		(en binaire)

et pour lesquelles l'exposant binaire varie entre -20 et + 20 rappel : (±4 * B).

Les nombres modèles au voisinage de 1 (c'est-à-dire $0.1 * 2^1$) sont donc :

30/32	31/32	1	1 + 2/32	1+4/32	1+6/32
0,9375	0,96875		1,0625	1,125	1,1875

`'EPSILON = 0,0625` (flèche vers 1,0625)

Les nombres modèles au voisinage de zéro :

$-17/32*2^{-20}$ $-16/32*2^{-20}$, 0, $+16/32*2^{-20}$ $+17/32\ 2^{-20}$

`'SMALL` (flèche vers $+16/32*2^{-20}$)

Les 3 nombres modèles les plus grands :

$29/32 * 2^{20}$; $30/32 * 2^{20}$ et $31/32 * 2^{20}$ soit 450272 ; 983040 et 1015808 (et ce sont des entiers !)

---

[1] 4.33E-19 = $0.1*2^{-60} = 2^{-61}$
[2] 1.15E+18 = $2^{60} - 2^{45}$
[3] c'est absurde concrètement mais imaginons cela comme exercice!

Problème d'approximation et conséquence [4]:

Si nous écrivons : `G : GROSSIER := 1.05;`
G sera converti en un nombre valant `1 ou 1 + 2/32 = 1.0625`
voir ci dessus (**mais nous ne savons pas lequel**)

Si nous calculons `G := G*G` on a
soit $1^2 = 1$
soit $(1 + 2/32)^2 = 1 + 4/32 + \varepsilon$ représenté par 1 + 4/32 ou 1 + 6/32

finalement $(1.05)^2$ qui doit valoir 1.1025 est représenté par 3 valeurs possibles :
soit par 1 soit par 1,125 (1+ 4/32) soit par 1,1875 (1 + 6/32)

« l'erreur » relative est respectivement en valeur absolue :
soit : 0.09 soit : 0.02 soit : 0.07

celle-ci est inférieure à 0.1. Elle "rentre" dans la précision demandée avec digit 1 **mais ceci n'est pas toujours le cas pour tous les calculs !**

En effet : Si `G := 1.06;`
on a une "erreur" relative égale à 0.11 pour G*G! refaire les mêmes calculs à titre d'exercice !

Réalisation (implémentation) avec les nombres meilleurs :

En pratique le type `GROSSIER` sera dérivé d'un type implémenté ayant plus de précision (heureusement !).
alors `type GROSSIER is digits 1;` est équivalent à

```
subtype GROSSIER is ANONYME;
```
où : `ANONYME` est le type prédéfini **choisi par le compilateur**

exemple avec Gnat 12p (AdaGide) on trouve :

digits 4 ⇒ 24 bits de mantisse (de 1 à 6)
digits 9 ⇒ 53 bits de mantisse (de 7 à 15)
digits 18 ⇒ 64 bits de mantisse (de 16 à 18)

Les attributs sur les types flottants ou digits sont nombreux on verra (en plus de ceux déjà signalés) notamment les pages notées « Attributs II » fichier attibut2.doc avec :

`'DIGITS, 'FLOOR, 'FRACTION, 'REMAINDER, 'ROUNDING, 'TRUNCATION.`
Pour plus de détails, encore, voir le manuel de référence A.5.3 (avec Ada Gide par exemple).

Paquetage `Ada.Numerics.`

Ce paquetage définit seulement deux constantes point flottant PI = 3.14……et e = 2.718…… par contre ses fils sont intéressants notamment le générique `Ada.Numerics.Generic_Elementary_Functions` qui propose toutes les fonctions mathématiques « usuelles » telles que : `SQRT`, `COS`, `SIN`, `TAN`, `LOG`, etc. Il existe aussi une série de lignée « fils » `Numerics.Generic_Complex_`….. dédiés aux … complexes !

On utilisera le paquetage `Ada.Numerics.Generic_Elementary_Functions` en TP.

[4] On continue à imaginer que le contrat digit 1 est réalisé exactement!

Type point-fixe : (`delta et delta-digits`) [5]

On distingue les types points fixes normaux (ou binaires) et les types points fixes décimaux. Les types point-fixe (binaires ou décimaux) correspondent à un nombre de chiffres après la virgule fixé **dû à une précision absolue** qu'on indiquera pour déclarer le type (avec **`delta`**).

L'intervalle (**`range`**) **est obligatoire** pour les points fixes binaires (il était facultatif pour les flottants).
Exemple : `type T_COEF is delta 0.1 range -1.0..+1.0;`

Pour les points fixes décimaux le range est **facultatif** mais on introduit en plus la notion de chiffres significatifs :
Exemple : `type T_FRANC is delta 0.01 digits 11;`
Ce dernier type permet de travailler avec des nombres décimaux distants chacun de `0.01` et de valeur absolue maximum de `999999999,99`. Très utile pour les calculs comptables. Annexe F du manuel de référence.

Aperçu sur les points fixes binaires.

Soit la forme générale `type F is delta D range L..R;` Celle-ci déclare des valeurs dans l'intervalle de `L à R` avec une précision de `D` qui doit être positive et `D,L,R` doivent être réels et statiques.

Les nombres modèles sont :
d'une part 0.0 et d'autre part les nombres multiples de PAS où PAS : une puissance de 2 inférieure ou égale à D (**`delta`**) : c'est l'intervalle entre 2 nombres modèles. PAS **est choisi** par l'implémentation.
Voyons sur notre exemple : `type T_COEF is delta 0.1 range -1.0..1.0;`

Supposons PAS $\Rightarrow 2^{-4} = 0.0625 = 1/16 < 0.1$ (puissance de 2 inférieure ou égale à 0.1 (minimum!) mais le compilateur peut choisir plus fin 2^{-5} ou plus). Dans notre exemple le nombre de bits de mantisse pour coder les nombres modèles est égale à 4. Ce 4 **n'a rien à voir** avec le 4 en exposant du 2^{-4} ci-dessus. Voyons pourquoi.

Les nombres modèles de `T_COEF` sont de la forme `i/16` avec `i` = ± (1, 2, 3,....) puisqu'ils sont tous distants d'un intervalle identique et égal à PAS = 1/16. Les bornes de l'intervalle (**`range`**) ne doivent pas être distantes de plus de PAS d'un nombre modèle donc :
-1.0 que l'on peut écrire -16/16 permet à l'extrême limite un nombre modèle de -15/16. De même :
+1.0 que l'on peut écrire +16/16 permet à l'extrême limite un nombre modèle de + 15/16.

On voit donc ici que `i` va de -15 à +15. En binaire 15 se code 1111 ⇒ 4 bits de codage ⇒ B = 4 (cqfd).

Les nombres modèles sont donc :

-15/16	-14/16	-13/16...	-1/16,	0	1/16.....	...14/16	+15/16
-0,9375	-0,875	-0,8125...	-0,0625, 0		0,0625...	0,875	0,9375

Remarque :
-1.0 et 1.0 sont ici en dehors de l'intervalle des nombres modèles. Ils sont exactement à un pas d'un nombre modèle. Il arrive parfois que les bornes soient nettement **dans** l'ensemble des nombres modèles.

Exemple :
`type T_AUTRE is delta 0.1 range -1.0..1.2;` Notez le `1.2` au lieu de `1.0`

supposons le PAS inchangé (il dépend du **`delta`** `0.1`) c'est donc 1/16. On approche toujours `-1.0` avec l'expression -15/16. Mais pour approcher `+1.2` il faut aller jusqu'à 19/16 = 1,1875. Pour coder 19 qui s'écrit en binaire 10011 il faut 5 bits de mantisse donc B = 5 et de fait les nombres modèles varieront de -31/16 à +31/16 (car avec 5 bits de mantisse on sature en binaire à 11111 qui vaut 31 en décimal). Les bornes du **`range`** sont donc dans cet exemple-ci largement dans le champ des nombres modèles. Mais le **`range`** interdira d'en sortir : il est impossible d'accéder aux nombres modèles hors de l'intervalle `-1.0..1.2`.

[5] Si le temps le permet !

Le problème des entrées-sorties sur les réels "point-fixe" est, sans problème, réglé avec l'instanciation du paquetage `TEXT_IO.FIXED_IO`. On hérite alors des traditionnels `GET` et `PUT`.

Définition :

On appelle `SMALL` le plus petit nombre modèle positif. On remarque qu'il est alors exactement égal au PAS. C'est évidemment **l'approximation** choisie par le compilateur pour réaliser le **`delta`** demandé par le programmeur. Cette approximation n'est pas toujours du meilleur effet sur les résultats des calculs. Voyons cela :

Supposons que nous désirions utiliser un type réel "point fixe" pour des calculs "financiers", tel que :

```
type EN_EUROS is delta 0.01 range ……..;
```

On sait que PAS : une puissance de 2 inférieure ou égale à D (**`delta`**) on a : $2^{-7} = 0.0078125 < 0.01 < 2^{-6}$

donc $2^{-7} = 1/128 = 1/2^7$ est le PAS minimum (supposons le implémenté). Croyant compter les centimes vous compterez en fait les 1/128 ièmes d'euros . Attention aux arrondis. Le langage ne garantit donc une **représentation exacte** que si **`delta`** est une puissance de 2, ce qui n'est pas le cas ici. La solution pour réaliser des calculs exacts est d'utiliser la clause de représentation ci-dessous.

Clause de représentation (' SMALL):

Les nombres modèles ne sont pas nécessairement les multiples de la précision indiquée. Avec l'attribut `SMALL`, on peut connaître la précision utilisée (puisqu'on a vu que `SMALL` = PAS), **mais on peut aussi avec une déclaration spéciale forcer** le `SMALL` à une valeur particulière et exacte. Ces demandes expresses s'appellent en Ada des **clauses de représentation**. Elles peuvent être refusées par le compilateur. Mais, si elles "passent" à la compilation, elles sont garanties.

La solution de notre problème est la clause de représentation suivante :

```
for EN_EUROS'SMALL use 0.01;
```

Dans ces conditions les calculs seront exacts (mais tous les compilateurs ne l'acceptent pas!).

Opérateurs sur les points-fixes :

Les opérations `+ , - , * , /` et **`abs`** peuvent être effectuées sur des valeurs "point-fixe".

- Addition et soustraction ne peuvent s'effectuer que sur des valeurs de même type **`delta`** et retournent un résultat de ce type. Ces deux opérations sont exactes !
- `*` et `/` sont autorisés entre des types "point-fixe" différents le type du résultat est déterminé par le contexte. Si le résultat est utilisé dans une expression (par exemple comme opérande d'une autre opération) alors **cette valeur doit être convertie par une conversion de type**.

Exemple :

```
DEL : constant := 2.0 **(-15);
type T_FRAC is delta DEL range -1.0..+1.0;
```

Soit `F,G,H : T_FRAC;`

```
F : = F + G;        accepté
F : = F * G;        accepté
F : = 0.5 * G;      accepté
F : = F + 0.5;      accepté
F : = F * G * H;    refusé
```

mais

```
F : = T_FRAC(F*G)*H;    accepté
```

Types décimaux (`delta-digits`).

Ils sont utilisés dans les applications de gestion et peuvent proposer une belle alternative au prétendu incontournable langage pour la gestion : le `Cobol`.

Le problème des entrées-sorties sur les réels "point-fixe décimaux" est, sans problème, réglé avec l'instanciation du paquetage `TEXT_IO.DECIMAL_IO`. On hérite alors des traditionnels `GET` et `PUT`.

L'annexe spécialisée F propose, en plus, un paquetage `Ada.Decimal` et des fils pour manipuler ces nombres commerciaux (calculs et surtout édition à la `Cobol` : les fameux `picture` ! [6]).

En première approximation on retiendra que ces nombres « décimaux » sont des points fixes (d'où le **`delta`** de leur déclaration) à **intervalle constant** avec un nombre de chiffres décimaux significatifs (d'où le **`digits`** de leur déclaration).

Exemple : **`type`** `T_FRIC` **`is delta`** `0.01` **`digits`** `18 ;`

18 chiffres significatifs (partie décimale comprise) et écart de 0.01 (donc des centimes).

Les opérations sont celles des points fixes avec la particularité de convertir par arrondi vers zéro calculs intermédiaires inclus. Il existe (déjà dit) une annexe spécialisée (F) et des paquetages (`Ada.Decimal` …..) qui propose des outils plus fins pour coller à l'ancêtre et spécialisé langage de gestion (Cobol).

En général la base `Machine_Radix` est une puissance de 2 mais certaines implémentations travaillent avec 10 (ce sont les fameux DCB que les vieux informaticiens affectionnent.

Des compléments à ce chapitre seront proposés en fonction de notre expérience. Notamment une étude comparative entre le vieux Cobol et le nouvel Ada repeint avec l'annexe F pourra être menée.

Remarque :

Dans le cours n°4 (sur le type tableaux) il est précisé que l'on peut comparer entre eux des tableaux (ou des tranches de tableaux) unidimensionnels de tailles différentes **à condition que les composants soient de type discret (ce qui exclut les réels !).** Peut-on, maintenant, comprendre pourquoi ?

[6] ils y reviendront au Cobol ! Gag !

Cours 12 Ada le type access (2 heures)

Avant propos : Il s'agit, avec ce cours n°12, d'une première approche à la « programmation de bas niveau » (terme non péjoratif !). Mais attention les facilités offertes nous permettent de « jouer avec le feu » ! Adieu le génie logiciel ! Nous verrons cela, encore plus poussé, au cours n°14 chapitre « l'interfaçage avec le langage C ».

Organisation de la mémoire

Que ce soit sous les systèmes d'exploitation : MS/DOS, Windows (NT, 95, 98) ou UNIX (ou Linux), la mémoire (physique ou virtuelle) mise à disposition de l'utilisateur est organisée « schématiquement » comme l'indique la figure ci dessous, une partie de la mémoire est réservée au code du programme, une autre aux données **globales** du programme, puis ce qui reste (à l'utilisateur) est accessible par les deux extrémités : d'un côté elle représente ce qui est appelé la **pile d'exécution**, de l'autre côté le **tas (ou pool en Ada)**. Entre les deux, la mémoire inoccupée à l'instant *t*. Aucune de ces trois dernières zones n'a de taille fixe, seule la somme des trois (PILE + inutilisée + TAS) ne doit pas dépasser les limites de la machine.

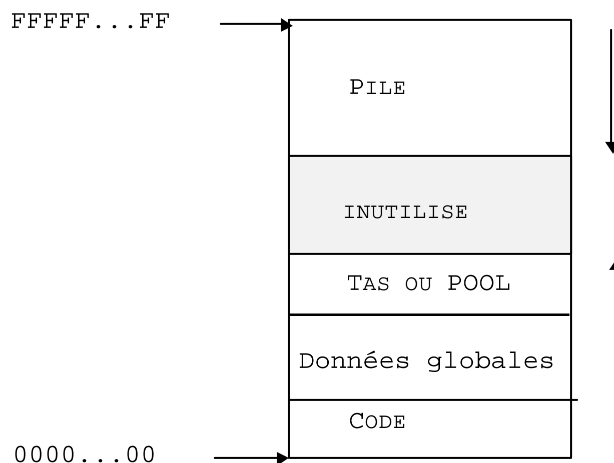


figure 1

Voyons les trois zones évoquées pour les données :

- la **zone de données globales** (ou données ***statiques***). Lorsqu'une variable est définie dans un paquetage, en dehors de tout sous-programme, le compilateur lui réserve le nombre d'octets nécessaires pour stocker les valeurs que peut prendre cette variable (suivant son type) : 1 ou 2 octets pour un caractère (ou Wide_Character), 1, 2, 4 (voire 8) octets pour un entier, etc... Cette zone se trouve à une certaine adresse mémoire (**supposons la fixée pour toute la durée du programme**) et l'identificateur de la variable statique est « assimilable » à cette adresse.

- la **pile d'exécution**. Lorsque l'appel d'un sous-programme est effectué, l'espace mémoire correspondant à ses variables locales (dites variables ***automatiques***) doit être réservé (voir cours récursivité notamment). Ceci est fait comme pour les variables dites statiques, mais a lieu dans la pile d'exécution, qui s'allonge alors au dépend de l'espace mémoire dit inutilisé. Cette zone sera réservée dans la pile jusqu'à la fin de l'exécution du sous-programme et sera alors « rendue » augmentant l'espace dit inutilisé ; les variables deviendront alors **logiquement** inaccessibles. Pendant toute la durée d'exécution d'un sous-programme, l'adresse mémoire correspondant à une variable locale sera fixée, et sera là encore « assimilable » à l'identificateur de la variable.

Ces deux cas de stockage d'objets (*statiques* ou *automatique*) présentent pour principal avantage de permettre un accès **direct -** donc rapide - à l'objet : **l'identificateur désigne directement l'objet**. En revanche, ils ont aussi de nombreux inconvénients :

- **gaspillage de l'espace mémoire :** prenons le cas d'un tableau : il doit être contraint avant de connaître le nombre effectif d'éléments qu'il aura à contenir pour un traitement particulier. Il est nécessaire de prévoir une taille suffisamment grande pour que le programme soit utilisable dans le plus grand nombre de cas

possibles. Si le programme est installé sur une autre machine disposant de plus (ou moins) de mémoire, soit le programme sera sous-dimensionné, soit il ne rentrera plus. Il faudra donc le modifier puis le recompiler. Si le programme est installé sur la même machine mais, doit tourner en « concurrence » avec des programmes résidents en nombre et en taille variables, il y aura là aussi un manque de souplesse. En conséquence, les déclarations statiques sont contraires au principe suivant : ***"une application ne devrait avoir pour seules limites que celles de la machine sur laquelle elle est utilisée"***. Le problème est semblable si l'application nécessite deux tableaux d'éléments différents, dont l'un peut être à un instant donné très plein et l'autre presque vide.

- **perte de temps :** la manipulation de variables de grande taille (occupation mémoire) représente des opérations coûteuses : la permutation de deux matrices de 100 x 100 réels par exemple de 4 octets par réel (et c'est un minimum) nécessite le mouvement de 120 000 octets.

- **objets de durée de vie imprévisible :** si une variable statique ou automatique devient obsolète en cours de déroulement du programme, il faut attendre la fin de l'exécution de la procédure (ou au moins du bloc) dans lequel elle est déclarée pour récupérer sa place mémoire.

- **objets de taille variable :** aucune solution simple n'est utilisable avec les variables statiques.

Les « pointeurs » : déclaration en Ada (`access`)

Pour pallier tous ces inconvénients (mais en ajouter d'autres !!!) les langages modernes (Pascal, C/C++, Ada, et les autres...), offrent un nouveau type d'objets : appelé type **pointeurs** (ou encore type référence). Remarquons en passant que Java (basé sur le concept de référence) ne permet pas de pointeurs pour l'utilisateur ! Et ceci pour des raisons de sécurité de développement (un bel exemple à garder en mémoire !). Le type de ces objets en Ada s'appelle le type **`access`**. Voyons son principe (première approche à compléter !).

Soit `T_Quelconque` un type quelconque préalablement défini (même un type fichier !). On déclare un type "pointeur sur un objet de type `T_Quelconque`" par :

```
type T_Ptr_T_Quelconque is access T_Quelconque ;
```

Notez bien le mot réservé **`access`**. Désormais, toute variable (ou instance) de ce nouveau type `T_Ptr_T_Quelconque` sera considérée comme pouvant « pointer » sur un objet de type `T_Quelconque`. On dit aussi référencer un objet de ce type `T_Quelconque`.

Par exemple :

```
Ptr_1, Ptr_2 : T_Ptr_T_Quelconque ; -- := null implicite !
```

On déclare ainsi deux « pointeurs » de ce type. Ils pourront « désigner » (mais indirectement) chacun un objet de type `T_Quelconque`. Notons que le "type pointeur" n'existe pas en soi, seul un type "pointeur sur un type donné" peut être défini. On retrouve les **exigences de fort typage de Ada**.

Création et accès à l'objet pointé

Ce n'est pas pour autant que `Ptr_1` et `Ptr_2` pointent sur quelque chose. Au contraire, ils ne pointent sur rien : leur valeur est **automatiquement** initialisée à **`null` à la déclaration.** Ces objets eux-mêmes sont des variables statiques !! Leur occupation est le nombre d'octets nécessaires pour stocker une adresse mémoire (ou ce qui en tient lieu). A tout moment l'utilisateur peut créer des objets **dans le corps d'un sous-programme** au moyen de l'opérateur (ou allocateur) **`new`** portant sur le type de l'objet qui doit être pointé. La valeur de retour est « en gros » l'adresse mémoire dynamique (dans le `TAS` ou `POOL`) que le système lui a allouée :

```
    ...
Ptr_1 := new T_Quelconque ;
    ...
Ptr_2 := new T_Quelconque ;
```

Les zones mémoire pour chaque objet pointé sont réservées mais elles n'ont toujours pas de valeur. Celles-ci vont pouvoir leur être données par les instructions désignant les objets pointés par le pointeur (notez bien le `.all`) :

```
Ptr_1.all := XXX ;
Ptr_2.all := YYY ;
```

La figure 2 résume la situation.

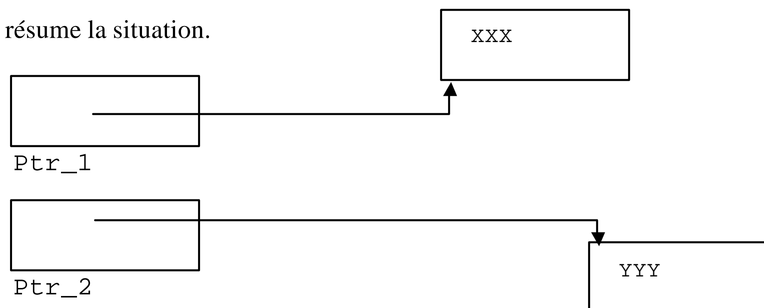


figure 2

La permutation des deux valeurs (qui ne changeront pas de place) sera obtenue par permutation des valeurs des deux pointeurs ainsi :

```
Ptr_3 := Ptr_1 ;
Ptr_1 := Ptr_2 ;
Ptr_2 := Ptr_3 ;
```

où `Ptr_3` est de même type que les deux autres pointeurs.
La figure 3 illustre la nouvelle situation.

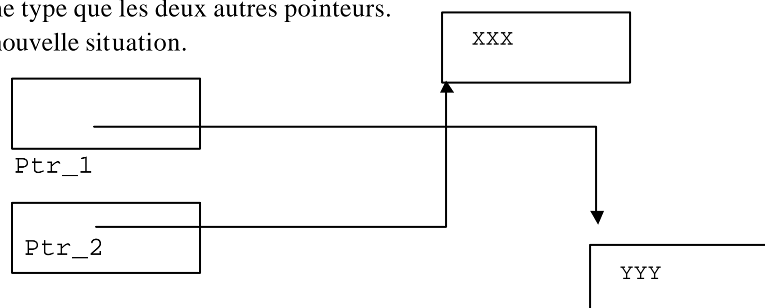


figure 3

On imagine l'intérêt qu'il y a à ne pas « bouger » la variable (surtout quand elle est très « volumineuse ») mais à manipuler uniquement son « référant ».

Applications : un tableur, un éditeur de texte, etc..., peuvent être construits selon ce principe. En mémoire **statique** figure un tableau de pointeurs vers des cellules pour le tableur, et un vecteur de pointeurs vers des chaînes pour l'éditeur (qui elles se trouvent en mémoire **dynamique**).

Remarques : La modification du pointeur `Ptr_1`, par exemple :

```
Ptr_1 := Ptr_2;
```

suffit pour que l'objet anciennement pointé par `Ptr_1` ne soit plus accessible (par ce pointeur au moins). Si aucun autre pointeur ne permet d'y accéder, il est inutilisable, mais pas pour autant détruit (figure 4). La place mémoire qu'il occupe est définitivement perdue **jusqu'à la fin de l'application.**

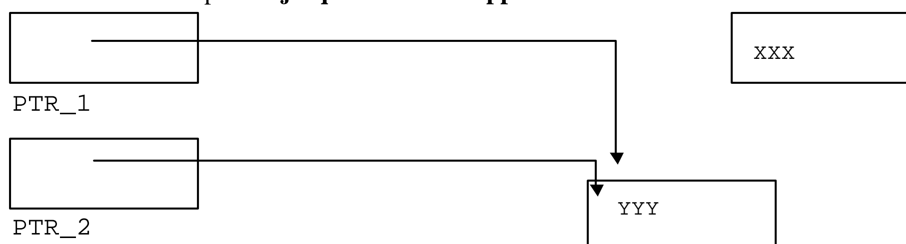


figure 4

Destruction de l'objet pointé (attention danger !)

La destruction d'un "objet" préalablement alloué en mémoire dynamique peut être obtenue par appel à une procédure provenant de l'instanciation de la procédure générique **`Unchecked_Deallocation`** paramétrée par le type du pointeur et celui de l'objet pointé. La destruction de l'objet pointé par `Ptr_1` par exemple peut être obtenue par :

```
procedure Dispose is new Unchecked_Deallocation (T_Quelconque,
T_Ptr_T_Quelconque) ;-- instanciation d'une procédure vraie
...
Dispose(Ptr_1) ; -- utilisation donc destruction !
```

Après appel à la procédure `Dispose`, la valeur de retour du pointeur `Ptr_1` est **`null`**.

A propos de la (dangereuse !) procédure générique `Unckecked_Deallocation` :

- elle se dénote en fait `Ada.Unchecked_Deallocation`, voir chapitre du manuel 13.11.2
- elle présente quelques risques. En effet, comme tout `Unchecked_...`, le contrôle exercé par le compilateur est supprimé et la responsabilité est transférée sur l'utilisateur. Si deux pointeurs pointent sur le même espace mémoire dynamique, et si, par l'appel de la procédure instanciée `Unckecked_Deallocation` sur l'un des deux pointeurs, l'utilisateur libère cet espace, la valeur de l'autre pointeur n'est pas modifiée mais elle est incohérente et toute tentative d'utilisation peut conduire à un comportement imprévisible du programme. En conséquence, cette utilisation doit être évitée le plus possible ou parfaitement contrôlée,
- le **pragma** `Controlled` doit être utilisé pour signaler au compilateur que le ramasse-miettes, s'il existe, (voir plus loin cette notion) ne doit pas gérer ce type de pointeur :

```
type T_Ptr_T_Quelconque is access T_Quelconque ;
pragma Controlled (T_Ptr_T_Quelconque);
```

Cependant des compléments propres à Ada permettent des techniques moins violentes (et recommandées) qui peuvent être pratiquées, certaines obligatoires, d'autres facultatives :

1. les objets pointés sont alloués sur le tas (déjà dit), mais une notion doit être introduite, celle de **pool de mémoire**. Lorsqu'un type accès est défini, un pool de mémoire lui est associé, qui peut être considéré comme un **tas spécifique**. Bien entendu, la réalisation pratique de ce pool n'a pas à être considérée. Lorsqu'on sort de la portée du type du pointeur, aucun pointeur de ce type ne peut plus exister, pas plus qu'un objet pointé. Ce pool est alors libéré. Par exemple :

```
declare
    type T_Ptr_Int is access Integer;
    Ptr_1, Ptr_2 : T_Ptr_Int;
begin
…………..
    Ptr_1 := new Integer;
    Ptr_2 := new Integer;
………………
end;
```

Remarque : à la fin de la portée du type `T_Ptr_Int` (le **end** du bloc **`declare`**) le pool mémoire est libéré ainsi que les objets qu'il contient (alloués mais non initialisés). Il faut noter de plus que, si deux types accès sont définis de façon identique, les objets pointés sont compatibles (l'ensemble des valeurs est le même) voir Barnes page 177 (au milieu). Ils peuvent par exemple être affectés l'un à l'autre. En revanche, les pointeurs sont complètement étrangers : un pointeur du premier type ne peut pointer sur un objet du second et inversement, ce qui est doublement normal : d'une part cette règle est déjà valable pour tout couple de type définis de façon identique (pensez à deux types différents de vecteurs de caractères), et d'autre part les objets pointés appartiennent à des pools de mémoire disjoints. Les problèmes que poserait la libération des objets seraient insurmontables lorsque les portées de ces deux types ne seraient pas identiques. Pas de conversion possible !

2. le noyau Ada **peut** posséder un **ramasse-miettes** (***garbage collector***). Il s'agit d'un mécanisme automatique permettant de récupérer l'espace de mémoire dynamique alloué, lorsqu'il devient inaccessible (voir figure 4). Ce mécanisme n'est malheureusement pas rendu obligatoire par la norme. Rappelons que, selon Meyer, un

langage à objets **doit** offrir un mécanisme **automatique** de récupération des objets qui deviennent inaccessibles. Cependant, il semble que certains problèmes posés par la réalisation des ramasse-miettes ne soit pas totalement résolus. Le principe habituellement utilisé dans ce type de situation est "le compteur de liens" : à chaque "objet"[1] alloué est associé un compteur de liens (un entier) qui est le nombre de pointeurs qui pointent sur cet "objet" à l'instant courant. A la création de "l'objet", le compteur est initialisé à 1, chaque affectation d'un pointeur sur cet "objet" l'incrémente d'une unité. Chaque suppression de ce pointeur ou réaffectation de ce pointeur sur un autre "objet" décrémente le compteur d'une unité. Lorsque le compteur de liens atteint la valeur nulle, "l'objet" devient inaccessible et peut donc être supprimé. Cependant certaines situations sont pratiquement insolubles : citons par exemple un pointeur sur un article dont un des champs est un pointeur sur l'article lui-même[2], ou, encore plus difficile à détecter, une suite de références circulaire. Le compteur de liens vaut donc 2. La destruction du pointeur (par sortie de sa portée par exemple) réduit le compteur à 1. "L'objet" ne peut donc pas être détruit mais est inaccessible. Seule une intervention du programmeur peut résoudre cette difficulté en détruisant préalablement le pointeur interne à "l'objet". Signalons que l'un des rôles des destructeurs d'objets[3] est justement de supprimer ces liens avant de détruire l'objet lui-même (procédure `Finalize` associée au type `Controlled` (cours Ada n° 10); voir aussi le cours de C++). A notre connaissance, aucun compilateur C++ n'offre non plus de ramasse-miettes intégré. Java par contre possède un ramasse miette intégré (voir dans le cours en question).

Constantes de type access - Parametre `in`

Il est bien entendu possible de définir des constantes de type `access`. Comme toutes les constantes, leur « valeur » doit être attribuée **au moment de la déclaration** au moyen de la qualification. Par exemple (de peu d'intérêt pratique!) :

```
type T_Ptr_Int is access Integer;
Cst_P_Cinq : constant T_Ptr_Int := new Integer'(5);--agrégat qualifié
```

Il est cependant intéressant de noter que c'est l'objet référence `Cst_P_Cinq` qui est constant et **non la valeur de l'objet référencé** correspondant. Après les instructions suivantes :

```
type T_Ptr_Int is access Integer;
Cst_P_Cinq : constant T_Ptr_Int := new Integer'(5); -- agrégat qualifié
Ptr_2      : T_Ptr_Int;
--...
begin
    Ptr_2 := new Integer'(7); -- agrégat qualifié
    -- équivalent aux deux instructions suivantes :
    -- Ptr_2 := new Integer;
    -- Ptr_2.all := 7;
```

l'instruction suivante est fausse (à cause du `constant` sur `Cst_P_Cinq`) :

```
    Cst_P_Cinq := Ptr_2;
```

tandis que l'instruction suivante est parfaitement valide :

```
    Cst_P_Cinq.all := Ptr_2.all;
```

Il est cependant fâcheux que la valeur pointée par `Cst_P_Cinq` soit maintenant égale à 7 !!! Cette remarque a une importance particulière avec les paramètres de type `access` qui sont des données (`in`) : le pointeur ne peut pas être modifié dans le corps du sous-programme (à cause du `in` !) , mais la valeur pointée peut l'être (prudence donc), comme dans l'exemple ci-dessous :

---

[1] Nous notons le mot "objet" entre guillemets, car il s'agit d'en prendre une acception plus générale que celle d'un langage à objets. Ce peut être ici un article, ou même un simple scalaire. Le mot "objet" désigne ici un espace mémoire contenant une information.

[2] On parle d'auto-référence

[3] Cette fois au sens des langages à objets !!!

```
procedure ChangeEntier (Ptr : in T_Ptr_Int) is
begin -- ChangeEntier
    Ptr.all := Ptr.all + 1;
end ChangeEntier;
```

Sans pouvoir appeler ce phénomène un ***effet de bord***, il est bon d'avoir toujours cette possibilité en tête, d'autant que l'identificateur ne rappellera pas (parfois volontairement) qu'il s'agit d'un type `access`. Danger !

Opérateurs applicables sur les type `access`.

Les valeurs d'un « pointeur » n'appartiennent pas à un ensemble ordonné. On sait (mais a-t-on besoin de le savoir) qu'il s'agit en gros de valeurs représentant des adresses mémoires (virtuelle). Mais rien n'indique qu'il s'agit par exemple d'entiers. Selon l'implémentation on peut même représenter une adresse sous forme de couple d'entiers (base + déplacement). En conséquence, les seules opérations de comparaison sur les pointeurs sont l'égalité et la non égalité. Le type pointeur est donc un type `private` ! En utilisant les déclarations ci-dessus, les instructions suivantes sont valides :

```
if (Ptr_2 /= Cst_P_Cinq)
then
    -- ...
end if;
```

ou

```
loop
    exit when Ptr_2 = Cst_P_Cinq;
    -- ... faire évoluer le pointeur Ptr_2
end loop;
```

Attention noter que cela signifie que la boucle cessera quand le pointeur `Ptr_2` pointera sur le même objet que `Cst_P_Cinq`, donc pas d'arrêt quand la valeur pointée par `Ptr_2` sera égale à 5.

En revanche, l'instruction ci-dessous est dénuée de sens, et syntaxiquement fausse :

```
if (Ptr_2 > Cst_P_Cinq)
then
    -- ...
end if;
```

Notons cependant que le paquetage `System` permet des opérations de comparaison (<, <=, >, >=, =, /=) sur des adresses Ada (type `Address` défini par l'implémentation) mais cela n'induit pas des opérateurs sur les pointeurs qui ne peuvent donc être assimilés de façon simpliste à des adresses !

Références circulaires, référence unidirectionnelle

Dans l'implémentation des structures de données classiques (listes, arbres, graphes, voir cours correspondant avec C++ [4]), il est fréquemment nécessaire de pouvoir passer d'un objet à un autre de même type. Il faut ajouter à chaque objet (de type article) un champ représentant une référence (un accès) à un autre objet. Les déclarations de type nécessitent de résoudre la difficulté suivante : l'objet contient un champ "pointeur sur objet" qui doit donc être déclaré préalablement. La définition de celui-ci nécessite à son tour le type "objet". Il s'agit d'une définition récursive qui peut être levée grâce à une déclaration partielle (ou retardée) du type "objet" :

```
type T_Personne;-- définition retardée
type T_Ptr_Personne is access T_Personne;
```

[4] On consultera aussi (en bibliothèque) un livre intéressant : Structures de données avec Ada95, Java et C++ de Christian Carrez.

```
type T_Personne is
    record
        Nom : T_Nom;
        -- ...
        Suivant : T_Ptr_Personne;
    end record;
```

Une utilisation possible en serait par exemple :

```
Ptr_1 : T_Ptr_Personne;
Ptr_2 : T_Ptr_Personne;
-- ...
begin
    Ptr_1 := new T_Ptr_Personne; -- création des objets pointés
    Ptr_2 := new T_Ptr_Personne;
    Ptr_1.all.Suivant := Ptr_2;  -- ou   Ptr_1.Suivant := Ptr_2;
    -- ...
```

La notation **`.all`** n'est pas toujours obligatoire (voir ci dessus l'abréviation en commentaire) mais fortement recommandée ! Dans cet exemple il faut aussi noter l'existence de deux pointeurs permettant d'accéder au second objet (`Ptr_2` et `Ptr_1.all.Suivant`), alors qu'il n'en existe qu'un seul pour accéder au premier (`Ptr_1`). Cette redondance peut conduire à des erreurs (voir plus haut le chapitre destruction) et devrait être évitée lorsque cela est possible : comme ici par exemple :

```
Ptr_1  : T_Ptr_Personne;
-- ...
begin
    Ptr_1         := new T_Ptr_Personne;
    Ptr_1.Suivant := new T_Ptr_Personne;
    -- ...
```

Rappelons que, avant l'instruction :

```
    Ptr_1.Suivant := new T_Ptr_Personne;
```

le champ `Suivant` de l'objet créé par le premier appel à l'opérateur **`new`** est initialisé **par défaut** à **`null`**. Il faut bien noter que le lien créé entre deux objets de type `T_Personne` est **unidirectionnel** : il est possible de passer du premier au second, mais impossible de remonter au premier. Cette dernière opération nécessiterait l'ajout d'un second champ :

```
type T_Personne is
    record
        Nom : T_Nom;
        -- ...
        Precedent  : T_Ptr_Personne;
        Suivant    : T_Ptr_Personne;
    end record;
```

Le chaînage complet ne peut se faire qu'en deux opérations :

```
    Ptr_1                    := new T_Ptr_Personne;
    Ptr_1.Suivant            := new T_Ptr_Personne;
    Ptr_1.Suivant.Precedent  := Ptr_1;
```

ce qui, reconnaissons-le, n'est pas d'une extrême élégance, mais cependant d'une parfaite efficacité !!! On a aussi (voir rappel précédent) `Ptr_1.Precedent` et `Ptr_1.Suivant.Suivant` qui valent **`null`**.

Type `access` et tableaux non contraints

Un type accès peut être défini sur des types quelconques et notamment non contraints (ici des tableaux). Mais au moment de la réservation de la mémoire par l'allocateur **`new`**, l'objet doit être contraint, car la taille maximum de l'objet doit évidemment être connue :

```
    type T_Vect_Car is array (Positive range <>) of Character;
    type T_Ptr_Vect_Car is access T_Vect_Car; -- ou access String !!
    Ptr  : T_Ptr_Vect_Car;
begin
    Ptr  := new T_Vect_Car (3..5);
    -- ...
```

Type `access` et articles avec discriminants

Comme pour les tableaux (contraints ou non), un type accès peut être défini sur un article (contraint ou non) avec discriminant. Par exe mple :

```
   type T_Chaine (Discri : Natural) is – voire mutable avec := 0 en plus
     record
        Vect : String (1..Discri);
     end record;
   type T_Ptr_Chaine is access T_Chaine;
   Ptr : T_Ptr_Chaine;

begin
    Ptr := new T_Chaine(12); -- contrainte obligatoire
    -- ...
```

Pour la même raison que précédemment, la valeur du discriminant doit être connue au moment de l'appel à l'allocateur de mémoire (même pour les articles a priori mutables !). **La contrainte porte évidemment sur l'objet alloué et non sur le pointeur**. Ainsi, il sera possible de faire pointer ultérieurement `Ptr` sur un objet contraint différemment. En revanche, il est impossible de modifier la contrainte de l'objet alloué, même s'il était mutable a priori (il est donc contraint à la déclaration). Par contre si le pointeur est d'un type pointeur sur un type contraint alors le pointeur ne pourra référencer d'autres objets contraints différemment (Barnes page 179).

Autres types `access`.

Pratiquement tous les types peuvent servir à définir un type **`access`**, **et même des types de fichiers**. (voyez le type `Stream_Access` et la fonction `Stream` pour « pointer » sur les fichiers de flots). Les pointeurs vers des types tâches (qui permettent de créer de tels objets dynamiquement) seront vus dans le cours correspondant (à venir rapidement). Il est aussi possible de définir des pointeurs sur des pointeurs. Quoique fort utile cette pratique semble beaucoup plus rarement utilisée que dans d'autres langages comme le C.

Nous allons maintenant voir des conditions d'utilisations « particulières » des types **`access`** (quatre cas) : le type **`access`** généralisé, le discriminant en tant que type **`access`**, le type **`access`** comme paramètre d'un sous- programme et le type **`access`** sur sous-programme. Certaines remarques se conjuguent comme le type **`access`** sur sous-programme comme paramètre d'un sous-programme (permettant de passer un sous-programme en paramètre d'un autre sous-programme).

Types `access` généralisés.

Cette notion offre la possibilité de pointer aussi sur des objets **statiques** (ceux de la pile d'exécution) alors que jusqu'à maintenant nous avons travaillé avec des objets pointés mais créés dans la zone des objets **dynamiques** (Tas ou Pool). Il est possible d'accéder aux constantes et aux variables de la pile grâce aux pointeurs généralisés. Voyons la mise en œuvre sur un exemple :

```
procedure Locale is

    type T_Ptr_Int_Dyn       is access          Integer;
    type T_Ptr_Int_Sta       is access all      Integer;
    type T_Ptr_Const_Int is access constant Integer;
    Var_I  : aliased          Integer;
    Cste_J : aliased constant Integer := 24;
```

```
    Ptr_I  : T_Ptr_Int_Sta;
    Ptr_J  : T_Ptr_Const_Int;

begin – Locale
    Var_I := 12;
    Ptr_I := Var_I'Access;
    Ptr_J := Cste_J'Access;
    -- ...
end Locale;
```

On a reconnu que le type `T_Ptr_Int_Dyn` est un type « pointeur » vers un entier dans la mémoire dynamique comme nous l'avons rencontré précédemment. Par contre les mots réservés **`access all`** et **`access constant`** indiquent que les deux types définis peuvent pointer sur toute variable ou constante entière, **quelle que soit sa localisation**.

Cependant, l'accès ne peut se faire que sur des objets (constantes ou variables) **explicitement** signalés de leur côté par le mot réservé **`aliased`**. La récupération de la référence de l'objet (son adresse mémoire pour fixer les idées) par le pointeur est obtenue grâce à l'attribut **`Access`** (l'allocateur **`new`** n'est **pas utilisé** puisque les objets statiques sont déjà alloués à leur élaboration). Sans précaution particulière, un pointeur vers un objet local à un bloc pourrait être récupéré en dehors de ce bloc et conduirait à de graves erreurs parfois difficiles à détecter et souvent dramatiques. Ce type **d'erreur est fréquent** en C/C++ qui n'assure aucun contrôle de cette sorte. Ada évite ce problème en imposant que **la durée de vie des variables/constantes pointées soit au moins égale à celle des types accès généralisés** correspondants : c'est la **règle d'accessibilité**. Il est cependant possible de passer outre ce contrôle au moyen de l'attribut `Unchecked_Access`, mais cela n'est évidemment pas conseillé.

Discriminants de type `access`.

Un discriminant d'un type article (et même d'un type tâche ou encore d'un type **`protected`**, à voir prochainement) peut aussi être d'un type **`access`**. Cette particularité permet aux objets instanciés d'être paramétrés par une autre structure avec laquelle ils sont associés. Notons toutefois que les types ayant un discriminant de type **`access` doivent être `limited`**, cette remarque ne s'applique donc qu'aux articles car il faut retenir, d'ores et déjà, que les tâches (**`task`**) et les **`protected`** sont **`limited`** par définition.

Le type **`access`** d'un discriminant peut être d'un type **`access`** nommé (c'est-à-dire avec un identificateur de type préalablement déclaré) ou être anonyme. On parle alors, quand il est anonyme, de discriminant **`access`**. Les discriminants d'un type **`access`** nommé se comportent comme les autres composants d'article (voir le cours n°6). Les discriminants **`access`** (ou anonyme) ont des propriétés semblables à celles des paramètres. Voyons un schéma :

```
type T_Donnees is ...... ;
type T_Article (Ptr_D : access T_Donnees) is limited
record
......
end record ;
```

Une déclaration d'un objet de type `T_Article` doit inclure un accès à un objet associé du type `T_Donnees`.

```
Mes_Donnees : aliased T_Donnees ;
Mon_Article : T_Article := (Mes_Donnees'Access,......);-- agrégat
```

Cas particulier et application : l'auto-référence. Considérons les deux déclarations :

```
type T_Externe ;
type T_Interne (Ptr : access T_Externe) is limited
record
......
end record ;
```

```
type T_Externe is limited
record
      Champ_1 : T_Interne (T_Externe'Access);
      .........
end record ;
```

le champ Champ_1 est de type T_Interne donc il possède un discriminant access dénoté Ptr qui se réfère à une instance de l'article T_Externe.

En déclarant ceci : OBJ : T_Externe ;

On a un objet a structure auto-référente : le premier champ de OBJ pointe sur OBJ lui-même et l'auto-référence est automatique ! Toutes les instances de T_Externe se réfèrent à elles-mêmes. On pourra éventuellement utiliser cette propriété pour gérer des structures complexes chaînées.

Paramètres formels de type access.

Un paramètre formel d'un sous-programme (fonction ou procédure) peut parfaitement être de type **access** à un type donné :

```
type T_Quelconque is ...
procedure Proc (Ptr : access T_Quelconque) is  ...
```

Implicitement le paramètre Ptr est de mode **in** (non indiqué), il est passé par valeur comme tout pointeur. Le paramètre effectif est tout objet compatible avec cette définition, par exemple :

```
type T_Ptr_Dyn   is access          T_Quelconque; -- accès à un pool
type T_Ptr_All   is access all      T_Quelconque; -- accès généralisé
type T_Ptr_Const is access constant T_Quelconque; -- accès généralisé
P    : T_Ptr_Dyn := new T_Quelconque;
Var  : aliased T_Quelconque;
Cst  : aliased constant T_Quelconque  := ...; -- une valeur de ce type

begin
..........
    Proc (Var'Access);
    Proc (Cst'Access);
    Proc (new  T_Quelconque);
    Proc (P);
    -- ...
```

Le paramètre effectif ne doit jamais avoir la valeur **null**, ce qui est vérifié automatiquement à l'appel. Dans le cas contraire, une exception Constraint_Error est levée. Par exemple, la séquence suivante provoque la levée de l'exception :

```
P : T_Ptr;
begin
....... rien ne portant sur P
    Proc (P);
```

puisque P est initialisé par défaut à **null**.

Bien entendu, la règle d'accessibilité énoncée plus haut reste valable. Compte tenu de la complexité que peuvent atteindre certains programmes (sous-programmes et blocs imbriqués, récursivité, etc.), l'énoncé de cette règle est lui-même complexe et sa vérification doit parfois être reportée au moment de l'exécution. En cas de non respect, l'exception Program_Error est levée.

Rappelons (ou signalons) aussi que les paramètres de sous-programme d'un type tagged sont par défaut aliased donc peuvent être référencés par un pointeur avec 'Access. Cours n°11 (fichier Stream).

Il faut remarquer que, comme précédemment, le pointeur passé en paramètre est une donnée, mais pas l'objet pointé. Il s'ensuit que l'objet pointé est « en fait » un paramètre **in out** : c'est la méthode systématiquement utilisée pour définir des paramètres données-résultats en langage C, et l'une des deux méthodes en C++ et malheureusement (pour certains Ada-tollahs !) possible en Ada. Le type de paramètre **access** contourne donc la rigueur des fonctions auxquelles sont interdits les paramètres **in out**. Sur ce point donc, Ada rejoint la souplesse de C/C++ mais perd de sa rigueur légendaire. Cette remarque est importante. D'ailleurs certaines applications à très haute sécurité (avionique par exemple) s'interdisent les pointeurs !

Accès à un sous-programme.

Problème : Considérons un paquetage permettant de gérer un ensemble (d'entiers pour fixer les idées; ce pourrait être un type générique). Imaginons qu'il offre la procédure `Parcourir` qui parcourt la totalité des éléments de l'ensemble. Nous désirons appliquer à chacun des éléments une opération, par l'intermédiaire de la procédure `Traitement` qui n'est pas déterminée au moment du développement du paquetage.

Solution par paquetage générique

La première solution consiste à construire un paquetage générique, dont le paramètre de généricité est la procédure `Traitement` :

```
generic
  with procedure Traitement (N : in Integer);
package P_Ensemble is
    procedure Parcourir;
    -- ...
end P_Ensemble;
```

Exemple du corps de la procédure `Parcourir` qui parcourt l'ensemble des 5 premiers entiers positifs...

```
package body P_Ensemble is
procedure Parcourir is
begin -- Parcourir
    for I in 1..5 loop
        Traitement (I);
    end loop;
end Parcourir;
-- ...
end P_Ensemble;
```

L'utilisation pourrait en être, dans un programme `Essai` :

```
with P_Ensemble;
with ADA.TEXT_IO; use  ADA.TEXT_IO;
procedure Essai is
  procedure Afficher (Nbre : in Integer);
  package P_Ensemble_New is new P_Ensemble (Afficher);
  procedure Afficher (Nbre : in Integer) is
  begin -- Afficher
    PUT_LINE (Integer'Image (Nbre));
  end Afficher;
begin -- Essai
    P_Ensemble_New.Parcourir;
end Essai;
```

Cette première solution, déjà vue au cours n° 9 sur la généricité, est parfaitement utile lorsqu'une seule version (instanciation) du sous-programme `Traitement` est nécessaire pour un ensemble donné. En revanche, si l'on désire appliquer successivement **plusieurs** traitements différents **sur le même ensemble**, cette solution est lourde puisqu'il faudrait instancier plusieurs fois le paquetage générique, ce qui donnerait plusieurs types d'ensembles distincts. Les différents traitements ne pourraient pas être appliqués **au même ensemble**.

Solution par sous-programme générique dans un paquetage non générique

Une autre solution, consiste à ne mettre en paramètre générique de la procédure `Parcourir`, que la procédure `Traitement`. Le paquetage n'étant plus générique :

```
package P_Ensemble is
  generic
    with procedure Traitement (N : in Integer);
    procedure Parcourir;
    -- ...
end P_Ensemble;
```

Nous pouvons, dans le même paquetage, instancier deux fois la procédure générique `Parcourir` par deux traitements différents :

```
with P_Ensemble; with ADA.TEXT_IO; use  ADA.TEXT_IO;
procedure Essai is
  procedure Afficher (Nbre : in Integer) is
  begin
    PUT_LINE (Integer'Image (Nbre));
  end Afficher;
  procedure Incrementer (Nbre : in Integer) is
  begin -- Incrementer
    PUT_LINE (Integer'Image (Nbre + 1));
  end Incrementer;
  procedure AfficherTout    is new P_Ensemble.Parcourir (Afficher);
  procedure IncrementerTout is new P_Ensemble.Parcourir (Incrementer);
begin -- Essai
    AfficherTout;
    IncrementerTout;
end Essai;
```

Ici, les deux traitements s'appliquent au même type ensemble.

Néanmoins, cette dernière solution manque de « souplesse » par rapport à la plupart des autres langages évolués, qui offrent des pointeurs sur sous-programmes (technique très largement utilisée par le C/C++) :

Solution accès à un sous-programme

Le problème ci-dessus est résolu de la façon suivante (la généricité n'est plus indispensable !) :

```
package P_Ensemble is
    type Ptr_Traitement is access procedure (N  : in Integer);
    procedure Parcourir (Traitement : Ptr_Traitement);
    -- ...
end P_Ensemble;
```

```
package body P_Ensemble is
   procedure Parcourir (Traitement : Ptr_Traitement) is
   begin
      for I in 1..5 loop
         Traitement(I);
      end loop;
   end Parcourir;
end P_Ensemble;
```

```
package P_Action is
   procedure Afficher (Nbre : Integer);
   procedure Incrementer (Nbre : Integer);
end P_Action;
```

```
with Ada.Text_Io; use Ada.Text_Io;
package body P_Action is
   procedure Afficher (Nbre : Integer)is
   begin
      Put_Line (Integer'Image (Nbre));
   end Afficher;
   procedure Incrementer (Nbre : Integer) is
   begin
      Put_Line(Integer'Image(Nbre+1));
   end Incrementer;
end P_Action;
```

et son utilisation est de la plus grande simplicité :

```
with P_Ensemble, P_Action;
use P_Ensemble, P_Action;
procedure Essai is
begin -- Essai
    Parcourir (Afficher'Access);
    Parcourir (Incrementer'Access);
end Essai;
```

Généricité ou pointeur ? Que choisir pour utiliser des sous-programmes ?

Soit à calculer l'intégrale : $\int f(x)dx$ (entre a et b) [5]

Nous verrons la solution avec générique (solution 1) et la solution avec pointeur (solution 2).

Déclaration de la solution 1 :

```
generic
     with function La_Fonction (X : in Float) return Float;
function Intégrer (A, B : in Float) return Float;
```

la fonction `Intégrer` réalisant le calcul est une **fonction générique**. La fonction à intégrer `La_Fonction` est **paramètre générique** de `Intégrer` [6].

Déclaration de la solution 2 :

```
type Ptr_Fonction is access function (X : in Float) return Float ;
......
function Intégrer (La_Fonction : Ptr_Fonction; A, B : Float) return Float;
```

la fonction `Intégrer` réalisant le calcul est une fonction **classique** (elle n'est pas générique) . La fonction à intégrer `La_Fonction` est **paramètre formel** de `Intégrer` (mais par le biais de son pointeur) en fait c'est `La_Fonction.all` qui est la fonction à intégrer.

[5] Cet exercice sera mis en œuvre en TP semaine 12
[6] remarquez au passage que Ada accepte les accents même dans les identificateurs !

Utilisation et mise en œuvre de la solution 1 :

```
function Integ_Cosinus is new Intégrer (Cos);
```

instanciation d'une vraie intégrale avec `Cos` comme fonction à intégrer.

```
.... := Integ_Cosinus (0.0, PI/2.0) ;
```

utilisation de `Integ_Cosinus` entre 0 et PI/2 puis affectation .

Utilisation et mise en œuvre de la solution 2 :

```
.... := Intégrer (Cos'Access, 0.0, PI/2.0) ;
```

utilisation de `Intégrer` avec la fonction `Cos` (en fait le pointeur sur `Cos`) entre 0 et PI/2 puis affectation .

Remarques :

- `PI` est défini dans le paquetage `Ada.Numerics` (voir le polycopié paquetages Ada)
- La fonction `Cos` est obtenu par instanciation ici avec le type `Float` du paquetage `Ada.Numerics.Generic_Elementary_Function`.
- Le codage de `Intégrer` sera vu en TP (déjà dit).

Que faut-il préférer de ces deux solutions ? (humour !) :

1. Pour les vétérans de Ada 83 (version qui ne possédait pas de pointeur) et qui ont pris l'habitude des génériques.
2. Pour les nouveaux programmeurs qui aiment les pointeurs et les retrouvent dans Ada95.

Plus sérieusement l'intérêt est dans le concept de portée [7]. Les pointeurs doivent avoir une portée plus grande que leur utilisation (car ils doivent exister au moment où on les utilise !!) sinon on risque une levée d'exception `Program_Error` . Cet inconvénient n'apparaît pas avec le générique qui est circonscrit à son instanciation. Il semble donc que pour des **raisons de sécurité** (ah le génie logiciel !) on peut encore garder ces bons vieux génériques ! Cependant quand on désire stocker l'identité d'un sous programme pour une utilisation ultérieure (problème classique du multi-fenêtrage par exemple) alors la **seule solution** passe par les pointeurs.

Type `access` et exceptions

Plusieurs exceptions peuvent être levées par le noyau Ada lors de la manipulation des pointeurs :

Storage_Error

Cette exception est levée lorsque le noyau Ada ne dispose plus de mémoire suffisante. Comme on le voit sur la figure 1, cet événement arrive lorsque la pile rejoint le pool (tas). Il y a donc deux cas possibles :

- la taille de la pile devient excessive. Rappelons que dans la pile sont placés les objets locaux des sous-programmes et des blocs (appelés objets **automatiques**), ainsi que les paramètres des sous-programmes. Une cause possible est une trop grande imbrication d'appels, peu probable, ou, plus probablement une récursivité incontrôlée (récursivité infinie). Une autre cause, très improbable, peut être l'empilement des valeurs intermédiaires lors de l'évaluation d'une expression très complexe.
- la taille du tas devient excessive. C'est le cas le plus fréquent. Cela ne signifie pas forcément qu'il s'agit d'une erreur, mais tout simplement que la limite de la machine est atteinte, soit parce que le nombre d'objets est trop important, soit parce que les objets sont trop volumineux. Il faut vérifier que la portée des pointeurs est la plus faible possible. Rappelons en effet que, en l'absence de ramasse-miettes, le pool de mémoire n'est récupéré que lorsque le contrôle du programme sort de la portée du type pointeur associé. Si le type accès est global, la mémoire n'est jamais récupérée. La seule solution est alors de prendre le contrôle de la libération de la mémoire par la procédure `Unchecked_Deallocation`.

[7] voir le livre de Rosen page 310.

Enfin, signalons que, l'utilisateur peut contrôler lui-même la gestion des pools de mémoire. Cet aspect est plus technique et réservé aux amoureux de la programmation système. Il existe cependant le paquetage `System.Storage_Pools` qui facilite cette gestion. Barnes 21.3 pages 542, 543. Manuel chapitre 13. En bref pour réserver de l'espace mémoire dans un pool on utilise l'attribut `'Storage_Size` voyons comment. L'espace mémoire d'un objet est fourni par l'attribut `'Size`, il renvoie le nombre de bits et on souhaite le nombre d'unités mémoire (en général l'octet) comment faire ? Il existe aussi la constante `Storage_Unit` (dans la paquetage `System`) qui renvoie le nombre de bits de l'unité mémoire donc le quotient `Objet'Size/System.Storage_Unit` résout notre premier problème et donne le nombre d'unités mémoire pour l'objet en question. Enfin pour réserver de l'espace (ici 50) :

```
for Ptr_Objet'Storage_Size use 50*Objet'Size/System.Storage_Unit ;
```

Constraint_Error

Cette exception est levée lors d'une tentative d'accès à un objet au moyen d'un pointeur invalide. Il s'agit essentiellement d'un pointeur dont la valeur est **`null`**. Comme souvent, l'utilisateur a la possibilité de laisser le noyau Ada détecter lui-même l'erreur et de capturer l'exception,

```
begin
    -- ...
    Ptr.all  := ...;  -- levée possible de Constraint_Error
    -- ...
exception
    when Constraint_Error => Traiter_Exception;
end;
```

ou de prévenir la levée possible de l'exception (moins coûteux !) :

```
begin
    -- ...
    if (Ptr = null)
    then
        Traiter_Exception;
    else
        Ptr.all  := ...;
        -- ...
   end if;
   -- ...
end;
```

Program_Error

Cette exception, levée dans différentes circonstances (par exemple sortie d'une fonction sans passage par une instruction `return`) est levée dans le contexte des types accès lors de conflits d'accessibilité.

Conclusion.

L'utilisation de la notion de pointeurs (très prisée dans les langages accédant aux ressources de bas niveau) est élégante mais extrêmement dangereuse en regard de la fiabilité d'un programme. L'implémentation du type `access` en Ada n'échappe pas à cette remarque. Cependant : le typage (indirect) du pointeur, la règle d'accessibilité, la non mixité des pointeurs et des adresses, etc. ajoutent une bonne dose de sécurité là où on est censé en perdre. Bien utilisée la notion de pointeur rendra service aux programmeurs qui cherchent à interfacer le langage Ada avec le langage C ou depuis peu avec le C++. A revoir cours n°14.

Rappel : Le langage Java conçu pour être plus sécurisé que C++ (son cousin) ne permet pas les pointeurs pour l'utilisateur. Cependant l'interprétation du code généré (byte code) utilise des pointeurs sur les objets (mais c'est transparent pour l'utilisateur).

Je retiens n°4

Quelques termes, informations ou conseils à retenir depuis les cours Ada n° 10, 11 et la suite….

Cette fiche fait évidemment suite aux fiches « je retiens n°1, n°2 et n°3 » vues précédemment.

Les fichiers :

- Textes, Séquentiels, Directs et Stream (flot de données) forment quatre classes bien distinctes.
- On retrouve dans les 4 classes de fichier des fonctionnalités identiques (`OPEN`, ….) et des identificateurs identiques (`FILE_TYPE`, `IN_FILE`, ….).
- Certains paquetages pour ces fichiers sont génériques d'autres non.
- Les entrées sorties ont des identificateurs différents (`GET`, `PUT`, `Read`, `Write`, voire même dénotés en attributs comme pour les Stream `'Read`, `'Write`, `'Input` ou `'Output`).

Les numériques :

- Certains sont portables (les entiers) d'autres presque portables (les réels au contrat minimum) à condition de spécifier le type à construire (types prédéfinis à bannir).
- On peut dériver un type numérique (**`tagged`** non nécessaire) pour accroître la sécurité de développement.
- Les réels flottants ont une précision relative (elle dépend du nombre), les réels fixes et décimaux ont une précision absolue (écart identique entre deux nombres quelconques).
- Les entrées sorties sur les numériques nécessitent l'instanciation des sous paquetages génériques de `Ada.Text_Io`.
- Se méfier des opérations relationnelles avec les types réels (notamment le =) ainsi que des pertes d'information dans les calculs. Sans oublier des dépassements possibles dans les évaluations intermédiaires. Voir cours numériques n°2.
- Les attributs essentiels sont à connaître ainsi que les paramètres de formatage pour les éditions.

Le type access :

- Une déclaration incomplète n'est utilisée qu'avec la notion de type **`access`**.
- On distingue trois sortes de type **`access`** : pour les données dynamiques, pour les données statiques et pour les sous programmes.
- Si une donnée accédée (ou pointée) possède un discriminant elle est toujours contrainte.
- Les variables pointeurs dynamiques ont une valeur initiale par défaut : **`null`**.
- Un allocateur muni d'une valeur initiale pour la donnée accédée utilise un agrégat qualifié.
- La donnée dynamique n'a pas d'identificateur propre elle se dénote grâce à son pointeur et au suffixe **`all`** s'il en est besoin.

Cours n° 2 les numériques (2 heures)

Les dangers et les pièges avec les réels (application à l'analyse numérique) :

Soit la séquence suivante (extraits) :

```
type T_D7 is digits 7; -- contrat de 7 chiffres significatifs

PAS    : constant T_D7   := 0.01;
S      : T_D7            := 1.0E6;
C      : LONG_ENTIER     := 0; -- grands entiers

.....
       loop
              S := S + PAS;
              C := C + 1;
              exit when S >= 1000001.0; -- il y a un os ici
       end loop;
```

Questions : A la sortie de la boucle quelle sera la valeur de la variable entière C ? Et d'ailleurs y-aura-t-il sortie de boucle ? Question pas si facile prenez garde !

A priori on a une trace (état à chaque itération) du style : (valeur de S, valeur de C)

(1000000.00, 0)	[(1000000.01, 1);	(1000000.02,2);..........	;(1000000.99,99)...].
état initial	itération 1	itération 2	;itération 99

A priori on **devrait** s'arrêter sur l'itération 100 ! **<u>Il n'en est rien</u>** ! **Pourquoi** ?

Ceci c'est toute la "magie" des réels "point-flottant" en machine où il conviendrait de s'intéresser de près aux notions de représentation mémoire (que fait la machine sur laquelle je travaille ?). Avec Ada, c'est plus facile car c'est au contrat (`digits` : nombre de chiffres significatifs minimum des nombres modèles) passé avec la machine qu'il convient de s'intéresser. C'est déjà mieux car les machines sont trop différentes entre elles (constructeurs, puissance, etc.).

Nous sommes donc dans un contexte de 7 chiffres significatifs et S vaut initialement 1000000 tandis que PAS vaut 0.01. <u>Tout deux ont bien 7 chiffres significatifs</u>. Faisons l'addition :

	1000000		
+	0000000.01		
=	1000000.01	$\Rightarrow$	le <u>résultat a plus de 7 chiffres significatifs</u> (il en a 9 !).

Il faut se souvenir que le compilateur **<u>peut s'en tenir au contrat demandé</u>**. En effet il a pu préparer, pour ses nombres implémentés un nombre de bits minimum pour représenter 7 chiffres décimaux (en fait 7*3.3219.. + 1 soit B = 25 bits) et si c'est le cas, la valeur du résultat à 9 chiffres peut ne pas être mémorisée (donc tronquée).

Cet algorithme est typiquement une faute de conception, et ce n'est pas la faute de la machine si l'addition risque de ne pas évoluer ! En réalité (ouf !) sur la plupart des machines les réels `digits` (nombre modèle) sont implémentés avec au moins 15 chiffres significatifs (à contrôler avec les attributs du type `'MODEL_`... (fichier attribut2.doc)). Donc l'algorithme évolue effectivement et ne se bloque pas. Et quand s'arrête-t-il ? C = 99 ? C = 100 ? C= 101 ?

Réponse : C = 101, àvérifier vous même ! Pourquoi ?

L'incrément 0.01 (qui vaut 1/100) s'écrit en binaire 0.000000101000111...... et il y a un «risque » non négligeable pour qu'il ne soit pas représenté exactement (cf. exercices faits en TD mantisse). L'incrément fait bouger la variable S mais d'un peu moins que 1/100 (eh oui ! c'est l'ordinateur !). On comprend pourquoi il faut une itération de plus (que le bon sens ne le prévoyait) pour atteindre un test de sortie positif. D'ailleurs que ferait le même programme avec une ligne ainsi :

```
exit when S = 1000001.0; -- attention on a = au lieu de >=
```

Réponse : Le test n'est jamais vrai ! car S à l'itération 100 est un peu inférieur à 1000001, puis à l'itération 101 la valeur est dépassée. **L'égalité stricte n'a aucun sens avec des réels**. L'algorithme se poursuit alors dans la boucle `loop` jusqu'à la levée de l'exception quand la variable entière C explose !

Remarque :

On ne doit jamais utiliser l'opérateur d'égalité (=) entre deux réels (points flottants).

Les calculs avec les réels :

La notion de perte d'information :

On se propose de faire le calcul de la suite : 1 + 1/2 + 1/3 + ... on arrête à une certaine BORNE puisque cette série est **divergente** (ne confondez pas avec $1 + ½ + ¼ + 1/8 + \ldots + 1/2^n + \ldots = 2$ cf. maths !).

Soit donc la variable BORNE valant le nombre de facteurs calculés. Il y a deux façons de procéder :

S_{mont} = 1 + 1/2 + 1/3 + ... + 1/BORNE

S_{desc} = 1/BORNE + ... + 1/2 + 1

Soit un algorithme de cette forme :

```
S_MONT := 0.0;
S_DESC := 0.0;
DIV_MONT := 1.0;
DIV_DESC := FLOAT(BORNE);
for I in 1..BORNE
loop
   INC_MONT := 1.0/DIV_MONT;
   INC_DESC := 1.0/DIV_DESC;
   S_MONT := S_MONT + INC_MONT;
   S_DESC := S_DESC + INC_DESC;
   DIV_MONT := DIV_MONT + 1.0;
   DIV_DESC := DIV_DESC - 1.0;
end loop;
```

On trouve (sur une machine, voir avec la vôtre !) les résultats suivants qui demandent réflexion :

pour 100 termes : 5.18737751763959 en montant
pour 100 termes : 5.18737751763961 en descendant

pour 1000 termes : 7.48547086054996 en montant
pour 1000 termes : 7.48547086055028 en descendant

pour 10000 termes : 9.78760603603648 en montant
pour 10000 termes : 9.78760603604363 en descendant

Remarques :

- le deuxième calcul (en descendant) est toujours supérieur au premier. Pourquoi ?
- une **même formule**, suivant la façon de procéder, **peut donner des résultats différents**.
- avec 10000 termes on est encore loin de l'infini (mais cette dernière remarque n'est pas de l'informatique).

Question : Quel est le calcul (montant ou descendant) le plus proche du résultat ? Etant entendu une fois pour toute que tout calcul informatisé n'est qu'une approximation.

Réponse : C'est la deuxième approche (évidemment la moins naturelle ! c'est toujours comme cela !) qui est la plus précise. Et la **solution a été « vue » précédemment**.

En effet la somme partielle `S_DESC` et son incrément se cumulent avec le moins de perte d'information puisque la somme partielle et l'incrément augmentent tout deux (peut être pas en proportion mais tout de même). Dans l'approche montante la somme partielle `S_MONT` augmente pendant que l'incrément diminue, on retrouve l'exercice n°1 et même avec des nombres implémentés forts précis il arrive un moment où l'incrément ajoute à la somme partielle moins qu'il ne devrait apporter (⇒ troncature).

Conclusion : il faut réfléchir, à la **notion de perte d'information**, quand on effectue des opérations arithmétiques entre deux termes (même pour des additions que l'on peut supposer a priori comme sans danger).

Retour sur les réels "point fixe" (`delta`) :

Soit l'extrait suivant d'une procédure :

```
type T_FIX is delta 0.1 range -20.0..+30.0;
NB : T_FIX := 0.4;
RES: T_FIX;

RES := NB + NB + NB;
```

Question : quel est le résultat naturellement attendu pour `RES` ?
Réponse : `1.2` ! Oui, bien sûr, et si ce n'était pas cela ? Voyons !

Détaillons en nous demandant si `0.4` est effectivement représenté par un nombre modèle de `T_FIX` ?

Quel est le `SMALL` minimum ? On a `1/16 < 0.1 < 1/8` le `SMALL` ou le PAS vaut au plus `1/16`.

Par quel encadrement est réalisé `0.4` ? On a `0.375 = 6/16 < 0.4 < 7/16 = 0.4375`

`0.4` est donc représenté en mémoire soit par `6/16` soit par `7/16`

`RES` est alors, à la fin, représenté soit par `1.125 (18/16)` soit par `1.3125 (21/16)`.
Sur nos machines c'est `1.125` qui est la valeur finale.

Question : les calculs sont-ils alors faux ?
Oui si la précision demandée n'est pas respectée. **Non** dans le cas contraire.

On a :
$(1.2 - 1.125)/1.2 = 0.0625 < 0.1$ d'une part
$(1.3125 - 1.2)/1.2 = 0.09375 < 0.1$ d'autre part

La précision (relative bien sûr) est respectée mais **avec les réels "point-fixe" il ne s'agit pas de précision relative !!!**

Voyons alors **la précision absolue** :
1.2 - 1.125 = 0.075 < 0.1 ⇒ oui
1.3125 - 1.2 = 0.1125 > 0.1 ⇒ non !!!

alors les **calculs sont faux** (par rapport à la précision souhaitée).

Question : comment forcer les résultats à être exacts ? Il faut « forcer » le `SMALL`
Réponse : avec la clause de représentation : `for T_FIX'SMALL use 0.1;`

Convergence théorique et convergence informatique

Introduction :

Pour exposer la problématique de ces notions de convergence (notamment le concept de test d'arrêt) nous allons tout au long de ce cours nous appuyer sur l'exemple représenté par la série suivante :

X = 4 * (1 - 1/3 + 1/5 - 1/7 + 1/9 .....)

le terme général dans la parenthèse est : $(-1)^n/(2n+1)$ pour n = 0, 1, 2, 3, 4 ....

Successivement on essaiera de calculer cette série en **factorisant différemment.** Deux exemples :

1°) X = 4 * [(1/1 - 1/3) + (1/5 - 1/7) + (...) ....] série croissante depuis 8/3
ou encore :
2°) X = 4 * [1 - (1/3 - 1/5) - (1/7 - 1/9) - (...) ...] série décroissante depuis 4

Remarque : les concepts vus pendant ce cours ont été (ou seront) intégralement appliqués dans le TP n°16 (dit « intégration ») qui suivra dans la semaine.

Première approche (série croissante) :

PROBLEME : Soit donc la série :

X = 4*(1/1 - 1/3) + 4*(1/5 -1/7) + 4*(1/9 -)......... infinie!

D'abord rappelons que l'infini est une illusion en informatique. Il faudra nous arrêter à un moment donné (donc choisir un test d'arrêt aux itérations clairement mises en évidence dans la formule).
Soit le programme `EXO1PRE1` qui réalise le calcul des 50 premiers termes de cette série (pour voir).

```
procedure EXO1PRE1 is
      type T_REEL is digits .....;
      X_NOUV      : T_REEL := 0.0;
      DIVIS       : T_REEL := 1.0;
      INCRE       : T_REEL;
begin
      for I in 1..50
      loop
            INCRE := 4.0/DIVIS - 4.0/(DIVIS + 2.0);
            X_NOUV:= X_NOUV + INCRE;
            DIVIS := DIVIS + 4.0;
      end loop;
end EXO1PRE1;
```

Les éditions successives des variables : I (numéro de l'itération), INCRE et X_NOUV donnent respectivement :

I	INCRE (ajout)	X_NOUV (valeur)	
1	2.66666666666667	2.66666666666667	
2	0.22857142857143	2.89523809523809	
3	0.08080808080808	2.97604617604618	
4	0.04102564102564	3.01707181707182	
5	0.02476780185759	3.04183961892940	
6	0.01656314699793	3.05840276592733	
7	0.01185185185185	3.07025461777918	
8	0.00889877641824	3.07915339419742	
9	0.00692640692641	3.08607980112383	
10	0.00554400554401	3.09162380666784	
11	0.00453771979580	3.09616152646364	
12	0.00378250591017	3.09994403237380	
13	0.00320128051220	3.10314531288601	
14	0.00274442538593	3.10588973827194	⇐ point intéressant !
15	0.00237882842700	3.10826856669894	
16	0.00208170699974	3.11035027369868	
17	0.00183696900115	3.11218724269983	
18	0.00163298632374	3.11382022902357	
19	0.00146118721461	3.11528141623818	
20	0.00131514055565	3.11659655679383	
21	0.00118994496505	3.11778650175887	
22	0.00108181203516	3.11886831379403	
23	0.00098777626868	3.11985609006271	
24	0.00090548953028	3.12076157959298	
25	0.00083307299802	3.12159465259101	
26	0.00076900893973	3.12236366153073	
27	0.00071206052514	3.12307572205588	
28	0.00066121167039	3.12373693372626	
29	0.00061562139284	3.12435255511911	
30	0.00057458880988	3.12492714392899	
31	0.00053752603642	3.12546466996541	
32	0.00050393700787	3.12596860697328	
33	0.00047340079295	3.12644200776623	
34	0.00044555834030	3.12688756610652	
35	0.00042010187470	3.12730766798123	
36	0.00039676635421	3.12770443433544	
37	0.00037532254281	3.12807975687825	
38	0.00035557135873	3.12843532823698	
39	0.00033733923677	3.12877266747375	
40	0.00032047430197	3.12909314177571	
41	0.00030484319628	3.12939798497199	
42	0.00029032843404	3.12968831340603	
43	0.00027682618776	3.12996513959379	
44	0.00026424442609	3.13022938401989	
45	0.00025250134141	3.13048188536130	
46	0.00024152401654	3.13072340937784	
47	0.00023124729007	3.13095465666791	
48	0.00022161278706	3.13117626945497	
49	0.00021256808822	3.13138883754319	
50	0.00020406601536	3.13159290355854	

Voici quelques idées ou questions pour animer la réflexion et rechercher une méthode de travail :

1. X tend-il vers une limite ? Au pif peut-être vers PI mais sait-on jamais ! (3.13159....≠ 3.14159....)
2. Quand arrêter les itérations ? (On se place dans le contexte où **on ne connaît pas la solution** et il faudra jouer le jeu jusqu'au bout). 50 itérations ?, 100 itérations ? ou plus encore ?
3. Remarquons que l'incrément au bout des 50 itérations est bien petit (0,0002....) en **valeur absolue**. Mais ceci ne veut rien dire : rappelez vous l'étude de la convergence informatique de la série 1 + 1/2 + 1/3 + ...1/N..... qui ne converge pas vers une limite (et pourtant l'incrément 1/N devient très petit).
4. L'incrément est déjà plus significatif si on le "prend" en valeur relative **c'est-à-dire ramené à la grandeur des variables calculées**. **La précision relative** est le seul critère acceptable (hélas non suffisant). En fait ce qui serait intéressant c'est de s'arrêter quand l'incrément est petit en valeur relative **par rapport à la valeur que l'on cherche**. Ici on peut observer la valeur relative de l'incrément par rapport à la valeur **que l'on calcule** (et ce n'est pas du tout pareil). **Ceci est très important !** Donc, à la 50-ième itération, la valeur relative de l'incrément ≈ 0,00006 est calculée avec 0,0002/3,13..... C'est plus intéressant que 0,0002 (disons plus significatif) mais ce n'est pas suffisant (voir question 6).
5. La série calculée est croissante puisque (1/n - 1/(n+2)) est positif. On peut aussi observer que X est majoré par 4 donc cette série converge (**c'est déjà cela !**). On peut montrer (voir cours de Math) que, effectivement, ça converge vers PI. Admettons le maintenant (ou de temps en temps) pour comprendre les erreurs d'interprétation que l'on peut commettre et les éviter à l'avenir.
6. Supposons que l'on veuille arrêter l'algorithme quand l'incrément est négligeable à 1 pour mille du **résultat calculé** (0.001 en valeur relative). On s'arrêterait alors (voir les éditions) à la quatorzième itération quand l'incrément est inférieur à 0.003. D'où une valeur estimée de 3,1058897 c'est minable non, comme approximation ? (on est critique quand on connaît la réponse !!).

Au fait à combien sommes nous en valeur relative du résultat PI (ignoré rappelez-vous !)

(3,14159....-3,10589.....)/3,14159..... = 0,01 environ ; c'est-à-dire à 1 pour cent (ce n'est plus du un pour mille !). Vous voyez déjà que les précisions (même relative) sur les résultats calculés n'ont rien à voir (souvent) avec la qualité de l'estimation des résultats souhaités !! Paradoxe de l'informatique dont il faut être toujours conscient.

Résumons :

On doit s'arrêter quand on est près du résultat (évident !), mais on ne connaît pas, en général, le résultat (sinon c'est un exercice gratuit). On peut s'arrêter quand le résultat calculé ne bouge presque plus en valeur relative (précision à fixer) mais **ceci ne permet pas de conclure vraiment** quant à la précision sur le résultat attendu.

Algorithme (faute de mieux !) :

La forme générale d'un algorithme « précision relative » est la suivante :

```
valeur_courante ⇐ valeur_initiale ;

répéter
        Calcul_nouvelle_valeur ;
        précision ⇐ (nouvelle_valeur - valeur_courante)/ nouvelle_valeur ;
        valeur_courante ⇐ nouvelle_valeur ;
jusqu'à abs(précision <= Précision_relative_souhaitée) ;
```

Mais rappelons *le peu de validité* de cet algorithme pris sans précaution !

Deuxième approche (série décroissante) :

Soit la même série observée différemment :

X = 4 + 4*(-1/3 + 1/5) + 4*(-1/7 + 1/9) + 4*........

Soit le programme `EXO2PRE1` qui réalise les 50 premiers termes de cette série (comme précédemment).

```
procedure EXO2PRE1 is

      type T_REEL is digits ....;

      X_NOUV      : T_REEL := 4.0;
      DIVIS       : T_REEL := 3.0;
      DECRE       : T_REEL;

begin
      for I in 1..50
      loop
            DECRE := 4.0/DIVIS - 4.0/(DIVIS + 2.0);
            X_NOUV:= X_NOUV - DECRE;
            DIVIS := DIVIS + 4.0;
      end loop;
end EXO2PRE1;
```

avec comme précédemment les éditions possibles suivantes (extraits) :

I	DECRE (retrait)	X_NOUV (valeur)
1	0.53333333333333	3.46666666666667
2	0.12698412698413	3.33968253968254
3	0.05594405594406	3.28373848373848
4	0.03137254901961	3.25236593471888
5	0.02005012531328	3.23231580940559
6	0.01391304347826	3.21840276592733
7	0.01021711366539	3.20818565226194
8	0.00782013685239	3.20036551540955
9	0.00617760617761	3.19418790923194
10	0.00500312695435	3.18918478227759
11	0.00413436692506	3.18505041535253
12	0.00347372991750	3.18157668543503
13	0.00295967443581	3.17861701099922
14	0.00255183413078	3.17606517686844
15	0.00222283967769	3.17384233719075
16	0.00195360195360	3.17188873523715
17	0.00173047804456	3.17015825719259
18	0.00154350762107	3.16861474957152
19	0.00138528138528	3.16722946818623
20	0.00125019534302	3.16597927284321
....		
30	0.00055559413848	3.15798499516866
....		
40	0.00031251220751	3.15393786227261
....		
50	0.00020000500013	3.15149340107098

On peut faire pratiquement les mêmes remarques que précédemment par exemple :
Cette fois la série est décroissante depuis 4, on sait aussi d'après le listing du premier programme que la série est bornée par 2,6666666.....donc elle converge, les tests d'arrêt (même en valeur relative) sont peu probants etc. Et finalement rien ne permet de conclure (pas plus que la première fois !)

Synthèse finale : (c'est finalement le seul point intéressant, **à retenir absolument** !)

Il en va autrement de la qualité de l'estimation **quand on conjugue pas à pas les deux algorithmes**.

On travaille **alors sur une série alternée** où il s'agit à chaque itération d'encadrer le résultat par une borne inférieure et une borne supérieure (résultats respectifs des itérations de l'algorithme n°1 et de l'algorithme n°2).

Intéressons nous à l'écart calculé entre deux itérations de notre série alternée. Quand cet écart est petit en valeur absolue mais aussi et surtout, comme on l'a vu, en valeur relative qu'en est-il ?

Entre deux itérations successives (l'une de l'approche n°1 (par défaut), l'autre de l'approche n°2 (par excès)) il y a forcément la valeur attendue (la vraie). Si l'écart entre deux itérations successives est petit (et toujours en valeur relative) alors **la valeur attendue est bien estimée par la moyenne arithmétique des deux itérations**. On peut même voir (à l'aide d'un dessin) que l'écart entre cette moyenne (l'estimation du résultat) et le résultat (inconnu), cet écart donc, est **inférieur** à l'écart entre les deux itérations. Donc l'écart entre deux calculs est plus grand que l'écart entre l'estimation de la valeur à calculer et celle ci. Intéressant !

Là est donc la solution, **dans un problème de convergence il faut toujours chercher à encadrer le résultat** et non pas tenter de l'approcher par un côté (quel qu'il soit !) c'est ce que faisait les deux premières tentatives (croissante et décroissante).

Voici un extrait d'une solution (sous forme de fonction pour changer).

A tester ! mais attention cela mouline plus !

```
begin
      DESC  : T_REEL := 1.0;
      MONT  : T_REEL := 1.0 - 1.0/3.0;
      DIVIS : T_REEL := 3.0;
      RES,
      DECRE,
      INCRE : T_REEL;

      loop
            DECRE := -1.0/DIVIS + 1.0/(DIVIS + 2.0);
            DESC  := DESC + DECRE;
            DIVIS := DIVIS + 2.0;
            --
            INCRE := 1.0/DIVIS - 1.0/(DIVIS + 2.0);
            MONT  := MONT + INCRE;
            DIVIS := DIVIS + 2.0;
            --
            RES   := (MONT + DESC)/2.0; -- moyenne
            --
            exit when abs((DESC - MONT) / RES) <= PRÉCISION;
            -- exit when abs(1.0/DIVIS / RES) <= PRÉCISION;
            -- car 1.0/DIVIS = DESC - MONT !
      end loop;
      return 4.0*RES;
end;
```

On a (sauf « erreur » à actualiser sur votre machine) les résultats suivants (avec un compilateur Ada83) :

3.1417091833... avec une précision relative demandée de 0.01
3.1415938839... avec une précision relative demandée de 0.001
3.1415926659... avec une précision relative demandée de 0.0001
3.1415926537... avec une précision relative demandée de 0.00001

Les calculs sont très, très longs si on devient gourmands !!!!

Autre point de vue :

On pourra aussi revenir à la définition originelle de la série et l'observer ainsi :

X = 4 * (1/1 - 1/3 + 1/5 - 1/7 .....)

Négligeant provisoirement le facteur 4 on a (dans la parenthèse) et terme à terme déjà les valeurs d'une série alternée (respectivement) :

1	1
1 -1/3 = 2/3	0.666666....7
2/3 + 1/5 = 13/15	0.866666....7
13/15 - 1/7 = 76/105	0.723809.....

etc.
On peut alors écrire une nouvelle version de notre fonction précédente (c'est presque la même mais les étapes sont plus brèves).

```
function CALCUL_PI (PRÉCISION : in T_REEL) return T_REEL is
      X_NOU : T_REEL := 0.0;
      X_OLD : T_REEL;
      NUMER : T_REEL := 4.0;
      DIVIS : T_REEL := 1.0;
      CREM  : T_REEL;
begin
      loop
            CREM  := NUMER/DIVIS;
            X_OLD := X_NOU;
            X_NOU := X_OLD + CREM;
      exit when abs(2.0*CREM / (X_OLD + X_NOU)) <= PRÉCISION;
            NUMER := - NUMER;
            DIVIS := DIVIS + 2.0;
      end loop;
      return (X_OLD + X_NOU)/2.0;
end CALCUL_PI;
```

Les résultats (à contrôler) sont les suivants :

0.01 ==> 3.141724829
0.001 ==> 3.1415938839
0.0001 ==> 3.1415926413
0.00001 ==> 3.1415926535

L'énigme de la semaine ? (de mal en PI):

Que	j'	aime	à	faire	connaître	un	nombre	utile	aux	sages
3	1	4	1	5	9	2	6	5	3	5

Troncature et perte d'informations dans les calculs

Introduction : On a déjà montré et mis en évidence la notion de perte d'information dans les calculs. Les conséquences de ces pièges informatiques étant souvent trop négligées nous allons revenir sur ces notions en guise de conclusion du module car c'est un problème crucial. Tout informaticien, digne de ce titre, face à des calculs programmés, se doit d'y être particulièrement vigilant. Pour illustrer les notions de troncature nous allons examiner trois exemples aussi divers que possibles et pris dans des domaines différents. A chaque fois nous montrerons le programme associé qui servira de base à la réflexion.

Exemple n°1 :

Soit 5 nombres **digits** `(NB(I), I = 1, 2, ..., 5)`. On calcule leur moyenne `MOY`. Puis on calcule la somme des carrés des écarts `(NB(I) - MOY)`^{2}. Ce calcul sert pour estimer la variance (cf. statistiques). Il y a deux méthodes :

a) méthode lourde, mais on applique bêtement la définition : $\sum$ `(NB(I) - MOY)`^{2}.
Soit `SC1` ce calcul.

b) méthode séduisante avec la formule <u>connue</u> en statistiques (démonstration facile !) :
[$\sum$`(NB(I)*NB(I))] - 5*(MOY*MOY)`.
Soit `SC2` ce calcul.

Il se trouve que les résultats `SC1` et `SC2` <u>ne sont pas les mêmes</u> pour deux séries de nombres ! Bien sûr les deux formules (on peut le montrer sont identiques). Voyons pourquoi les calculs diffèrent.

```
procedure EXO1TRO1 is

type T_REEL is digits ....;
type T_NB is array(POSITIVE range <>) of T_REEL;

NB1 : T_NB(1..5) :=(125.2642, 125.2638, 125.2650, 125.2629, 125.2644);
NB2 : T_NB(1..5) :=(125.2632642, 125.2632638, 125.2632650,
                       125.2632629, 125.2632644);
MOY : T_REEL;
S,SC1,SC2 : T_REEL := 0.0;

begin -- NB ci-dessous sera tour à tour NB1 puis NB2

-- calcul de la moyenne et début de SC2
      for I in NB'RANGE loop
            S := S + NB(I);
            SC2 := SC2 + NB(I) * NB(I);
      end loop;

      MOY := S/T_REEL(NB'LENGTH);

-- calcul de la somme des carrés des écarts
-- formule itérative associée à la définition

      for I in NB'RANGE loop
            SC1 := SC1 + (NB(I) - MOY)*(NB(I) - MOY);
      end loop;

-- calcul par la formule directe connue en statistique
      SC2 := SC2 - (S*S)/T_REEL(NB'LENGTH);
```

```
-- calcul(avec NB1 pour NB);
-- moyenne: 125.26406
-- SC1: 2.43199999999207 E-6
-- SC2: 2.43201024119344 E-6
--
-- calcul(avec NB2 pour NB);
-- moyenne: 125.26326406
-- SC1: 2.43199999999776 E-12
-- SC2: 3.49587025993469 E-12
end;
```

Quel est le meilleur calcul ?

Réflexions: (C'est un problème de troncature)

C'est `SC1` (résultat de la méthode dite lourde et non pas la plus concise !) et dans les deux essais qui est le plus près du résultat. En effet `NB(I) - MOY`, même élevé au carré, garde toujours un nombre de chiffres significatifs cohérent avec les nombres modèles. Si erreur il y avait elle proviendrait des représentations mémoire périodique (peu important).

En revanche pour `SC2` on cumule `NB(I)* NB(I)` qui perd forcément des chiffres significatifs surtout quand `NB(I)` est à … 10 chiffres !

Exemple n°2 :

Il s'agit une fois de plus d'étudier un algorithme qui « prétend » calculer `PI`.

Soit un cercle de rayon 1. Soit l'hexagone régulier inscrit dans ce cercle. Ses côtés (6 pour mémoire !) sont les bases de 6 triangles équilatéraux (de longueur : rayon du cercle). Le demi périmètre de cet hexagone vaut alors 3. On notera P_n le demi périmètre d'un polygone à n côtés ici donc $P_6 = 3$.

Il existe une formule mathématique qui donne le demi périmètre du polygone inscrit qui a deux fois plus de côtés (P_{2n}) en fonction du nombre de côtés N et du demi périmètre initial P_n.

Cette formule est (`SQRT` veut dire racine carrée) :

$$P_{2n} = SQRT(\ 2*N * (N - SQRT(N*N - P_n*P_n)))$$

Voir bons ouvrages pour la démonstration que l'on acceptera.

exemple : (demi-périmètre du dodécagone régulier)

$$P_{12} = SQRT(12*(6 - SQRT(36-9))) = 3.10582854......$$

Il est clair que plus le nombre de côtés augmente plus on tend vers le 1/2 périmètre du cercle de rayon 1 qui vaut ….. `PI`. Formule séduisante que l'on peut programmer (voir aussi la sortie écran qui est en commentaire après le programme page suivante). Catastrophe à la 14-ième itération alors que l'on convergeait bien vers `PI` (édité en fin de programme pour mémoire) les plombs sautent et c'est sans retour !

```
with ADA.TEXT_IO; use ADA.TEXT_IO;
with Ada.Numerics.Generic_Elementary_Functions; use Ada.Numerics ;
procedure EXO2TRO1 is
package IO_FLOAT is new FLOAT_IO(FLOAT);
use IO_FLOAT;
package P_FONCTION is new Generic_Elementary_Functions(Float) ;
use P_FONCTION ;
      N: FLOAT := 6.0;
      RES : FLOAT := 3.0;
begin
      for I in 1..24 loop
            RES := SQRT(2.0*N*(N - SQRT (N*N - RES*RES)));
            N   := N + N;
            PUT ("--itération: ");
            PUT(INTEGER'IMAGE(I));
            PUT (" Nomb de côtés: ");
            PUT(N,10,0,0);
            PUT("   Résultat: ");
            PUT (RES,2,14,0);
            NEW_LINE;
      end loop;
      PUT (PI,15,14,0);
end EXO2TRO1;
```

```
--itération:  1 Nomb de côtés:       12.0   Résultat:  3.10582854123025
--itération:  2 Nomb de côtés:       24.0   Résultat:  3.13262861328124
--itération:  3 Nomb de côtés:       48.0   Résultat:  3.13935020304686
--itération:  4 Nomb de côtés:       96.0   Résultat:  3.14103195089050
--itération:  5 Nomb de côtés:      192.0   Résultat:  3.14145247228534
--itération:  6 Nomb de côtés:      384.0   Résultat:  3.14155760791141
--itération:  7 Nomb de côtés:      768.0   Résultat:  3.14158389214980
--itération:  8 Nomb de côtés:     1536.0   Résultat:  3.14159046323329
--itération:  9 Nomb de côtés:     3072.0   Résultat:  3.14159210600136
--itération: 10 Nomb de côtés:     6144.0   Résultat:  3.14159251681049
--itération: 11 Nomb de côtés:    12288.0   Résultat:  3.14159262019715
--itération: 12 Nomb de côtés:    24576.0   Résultat:  3.14159264976795
--itération: 13 Nomb de côtés:    49152.0   Résultat:  3.14159264887861
--itération: 14 Nomb de côtés:    98304.0   Résultat:  3.14159265955078
--itération: 15 Nomb de côtés:   196608.0   Résultat:  3.14159270223946
--itération: 16 Nomb de côtés:   393216.0   Résultat:  3.14159253148473
--itération: 17 Nomb de côtés:   786432.0   Résultat:  3.14159412519519
--itération: 18 Nomb de côtés:  1572864.0   Résultat:  3.14159412519519
--itération: 19 Nomb de côtés:  3145728.0   Résultat:  3.14157226852405
--itération: 20 Nomb de côtés:  6291456.0   Résultat:  3.14155769732548
--itération: 21 Nomb de côtés: 12582912.0   Résultat:  3.14120796828227
--itération: 22 Nomb de côtés: 25165824.0   Résultat:  3.14493640635228
--itération: 23 Nomb de côtés: 50331648.0   Résultat:  3.15238005322962
--itération: 24 Nomb de côtés:100663296.0   Résultat:  3.15238005322962

-- valeur de Ada.Numerics.PI :
-- 3.14159_26535_89793_23846_26433_83279_50288_41971_69399_37511
```

Explication : `N*N - Pn*Pn` tend de plus en plus vers `N*N` car `Pn*Pn` (≈ 9,….) ne soustrait plus rien au premier opérande `N*N` qui, lui, augmente sans cesse et le résultat diverge alors finalement et sûrement vers .... zéro ! D'ailleurs il ne faut pas pousser beaucoup plus loin les itérations. Essayez !

Exercice n°3 :

On cherche à calculer `SQRT` (racine carrée). Cette fonction est disponible avec le paquetage `Ada.Numerics`……. mais bon, essayons de faire le calcul par la méthode dite de Newton-Raphson (voir cours de math : recherche du zéro d'une fonction). Rappelons brièvement la formule et la théorie :

Soit à calculer `SQRT(A)` avec `A >= 0`. On pose $F(x) = A - x^2$. On voit aisément que `X` solution de `F(X) = 0` est alors tel que $A = X^2$ donc `X = SQRT(A)` quand `F(X) = 0`..

L'algorithme (ie. recherche de `X` tel que `F(X) = 0`) est le suivant : on part d'une valeur X_0 (il faut la choisir sérieusement!) puis on fait la récurrence : $X_{n+1} \Leftarrow X_n - F(X_n)/F'(X_n)$ (formule apparentée au point fixe). On verra peut-être en TP une application de cet algorithme.

En posant $F(x) = A - x^2$ comme annoncé

On a : $F'(x) = -2x$

et $F(x)/F'(x) = (A - x^2)/(-2x)$

enfin $x - F(x)/F'(x) = (x + A/x)/2$

d'où il vient la récurrence : $X_{n+1} \Leftarrow (X_n + A/X_n)/2$

avec X_0 = A/2 (ou d'ailleurs toute initialisation positive car dans cet exemple l'algorithme est robuste)

Cette formule se programme aisément (deux versions ci-dessous). Le problème est celui du test d'arrêt. Précision relative cela s'impose mais entre les deux solutions proposées quelle est la meilleure ?

```
function RACINE1 (A: in FLOAT) return FLOAT is
      RES          : FLOAT := A/2.0;
      PREC_CAL     : FLOAT;
begin
      if A <= 0.0
      then return 0.0;
      end if;
      loop
            RES := (RES + A/RES) / 2.0;
            PREC_CAL := abs(A - RES*RES) / A;
      exit when PREC_CAL <= PRÉCISION;
      end loop;
      return RES;
end RACINE1;

function RACINE2 (A: in FLOAT) return FLOAT is
      RES          : FLOAT := A/2.0;
      COUR,
      PREC_CAL     : FLOAT;
begin
      if A <= 0.0
      then return 0.0;
      end if;
      loop
            COUR := RES;
            RES := (RES + A/RES) / 2.0;
            PREC_CAL := abs(RES - COUR)/RES;
      exit when PREC_CAL <= PRÉCISION;
      end loop;
      return RES;
end RACINE2;
```

Voir l'édition des résultats avec `A = 365.78`. La méthode 2 est très proche du résultat avec une précision non poussée alors qu'il faut une précision fine pour que la méthode 1 soit aussi proche du résultat. Bizarre ! Qu'en pensez-vous ?

Extrait de copies d'écran :
entrez le nombre: 365.78

Précision? (0.0 = sortie): 0.1
méthode 1: 19.17171059093580
méthode 2: 19.17171059093580

Précision? (0.0 = sortie): 0.01
méthode 1: 19.17171059093580
méthode 2: 19.12543180495620

Précision? (0.0 = sortie): 0.001
méthode 1: 19.12543180495620
méthode 2: 19.12537581338300

Précision? (0.0 = sortie): 0.001
méthode 1: 19.12543180495620
méthode 2: 19.12537581338300

Précision? (0.0 = sortie): 0.0001
méthode 1: 19.12543180495620
méthode 2: 19.12537581338300

Précis ion? (0.0 = sortie): 0.00001
méthode 1: 19.12537581338300
méthode 2: 19.12537581330100

Précision? (0.0 = sortie): 0.0
méthode SQRT: 19.12537581330100

Un petit dernier :

Voici un petit morceau de programme à tester vous même.

```
type T_NB is array (1..10) of FLOAT;
NB : T_NB := (-1_355_100.0, 287_312_000_000.0, 15_185.0,
                  -144_239_000_000.0, 444_998.0, -425_187.0, 24_998.0,
            4_972.29, -143_072_000_000.0, 1_417.49);
RES : FLOAT := 0.0;
begin
   for I in 1..10
   loop
      RES := RES + NB(I);
   end loop;
   TEXT_IO.PUT_LINE(" édite -288_716.219_991_145_5");
   TEXT_IO.PUT_LINE(" au lieu de -288_716.22 pour RES");
   TEXT_IO.PUT_LINE(" il y a problème a l'itération 9 ");
   TEXT_IO.PUT_LINE(" pourtant: 143072000000.0 - 143071709866.29");
   TEXT_IO.PUT_LINE(" donne quand il est fait seul: - 290133.71");
   TEXT_IO.PUT_LINE(" et a l'itération 9 c'est: -290133.709991455");
   TEXT_IO.NEW_LINE;
……
end ;
```

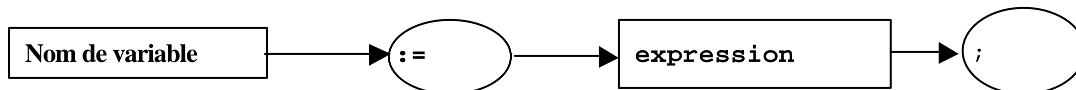
Cours 13 (2 heures : 1h expression, 1h dérécursification)

Les expressions.

Le concept **d'expression** est cité partout en Ada dans les définitions syntaxiques (voir par exemple les premiers diagrammes syntaxiques cours n°1 ou toute la définition du langage en B.N.F. dans AdaGide chapitre 4.4 ou dans le M.R.L.). Il est temps, avant de finir le cours Ada, de comprendre ce concept **jamais vraiment défini**. Ce sera l'occasion de présenter enfin les **priorités** et de revoir les D.S. (bouclant ainsi sur le cours n°1 où les D.S. étaient en bonne place). Attention ce cours peut être avancé dans le temps pour des raisons pédagogiques.

Exemple de définition où expression est utilisée (avec D.S.) :

Instruction d'affectation :



Exemple (le même mais en B.N.F.) :

```
assignment_statement ::= variable_name := expression;
```

Une expression en Ada est constituée hiérarchiquement :

- **de primaires,**
- **de facteurs,**
- **de termes,**
- **d'expressions simples**
- **de relations**.

Nous avons donc 6 définitions à modéliser: primaire, facteur, terme, expression simple, relation et enfin ne pas oublier expression elle même ! Nous le ferons dans cet ordre.

Primaire :

On remarquera (cf. D.S. ci dessous) que cette première brique (elle même la plus élémentaire) est formée (au choix) de 11 entités (donc au même niveau). Parmi ces entités on trouve …. expression (entre parenthèses) que l'assemblage des briques étudiées va permettre de fabriquer. Quel bel exemple de définition récursive qui ne vous échappera pas (voir les TD récursivité déjà étudiés !).

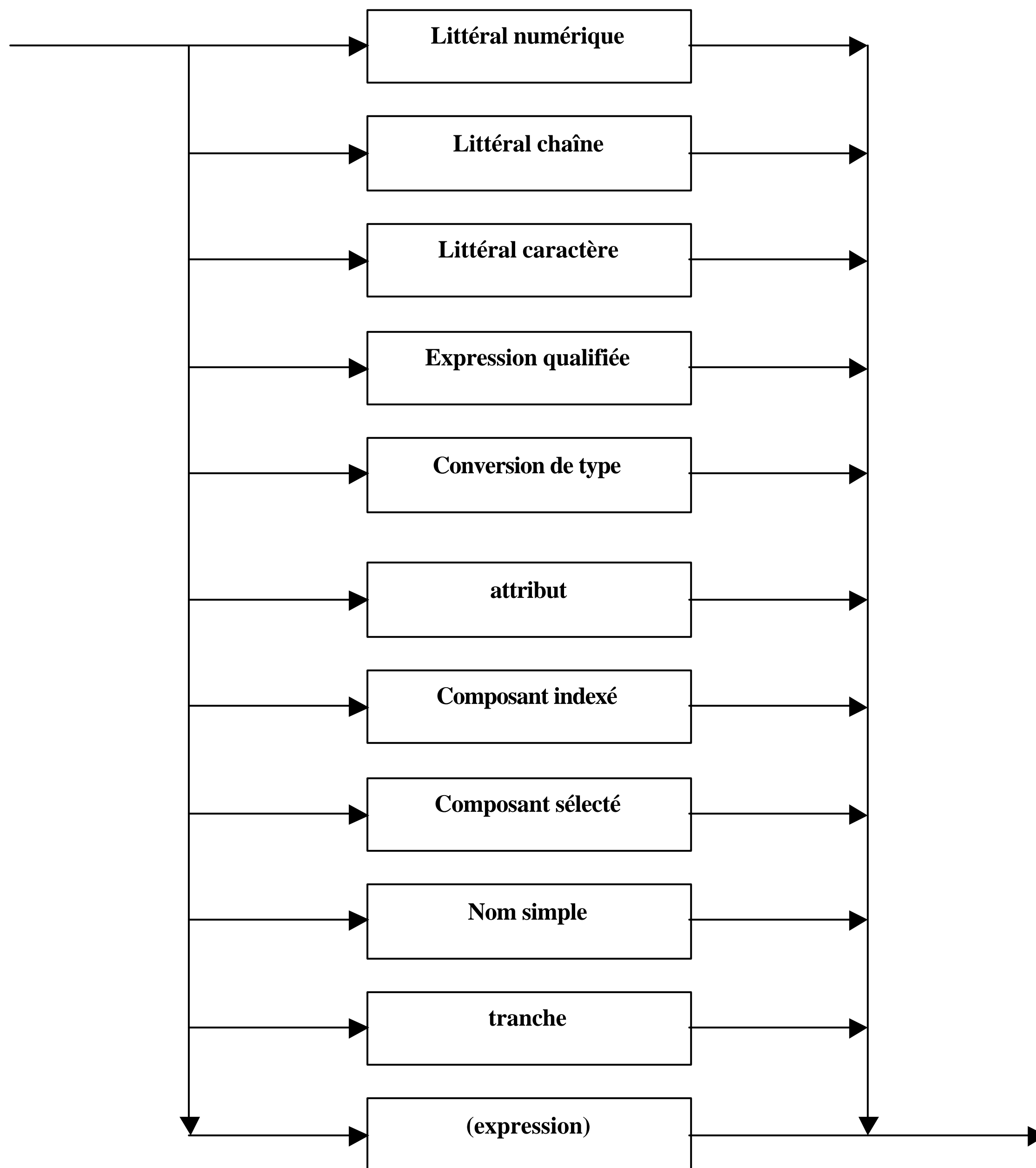
Exemples de primaires (dans l'ordre et respectivement) :

- Littéraux: `26.2` : **numérique** ;
- `"bonjour"` : **chaîne** ;
- `'Z'` : **caractère** ;
- `T_PEINTURE'(BLANC)` : **expression qualifiée** ;
- `T_ENTIER(2.6)` : **conversion de type** ;
- `T_PEINTURE'FIRST` : **attribut** ;
- `VECT(4)` : **composant indexé** ;
- `LA_DATE.JOUR` : **composant sélecté** ;
- `IND` : **nom simple** ;
- `LIGNE(4..7)` : **tranche**.

Il manque (voir page suivante) un exemple de primaire en tant qu'expression entre parenthèses.

Primaire (D.S.) :

Notez bien le dernier pavé !!



ó **Copyright D. Feneuille I.U.T. Aix 2002** (cours n°13 fichier Cours13.DOC)

Facteur :

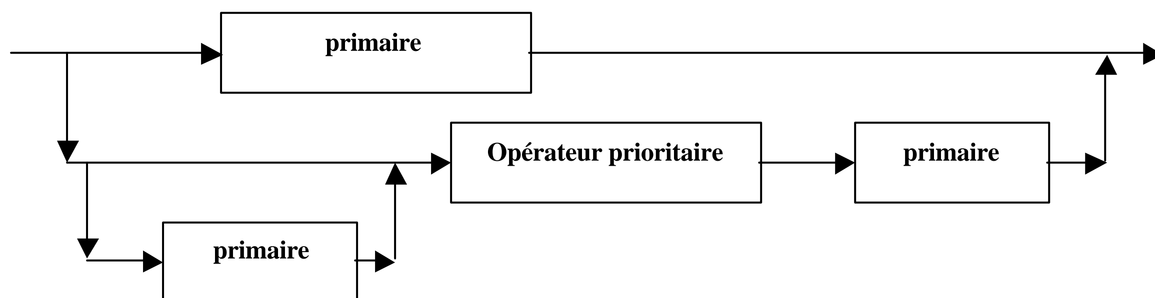
On remarque (voir diagramme ci-dessous) que tout primaire peut être lui même considéré comme un facteur. L'assemblage de deux primaires avec l'opérateur (binaire) prioritaire `**` ou la composition d'un opérateur (unaire) prioritaire (**`abs`** ou **`not`**) avec un primaire forme aussi un facteur.

Exemples :

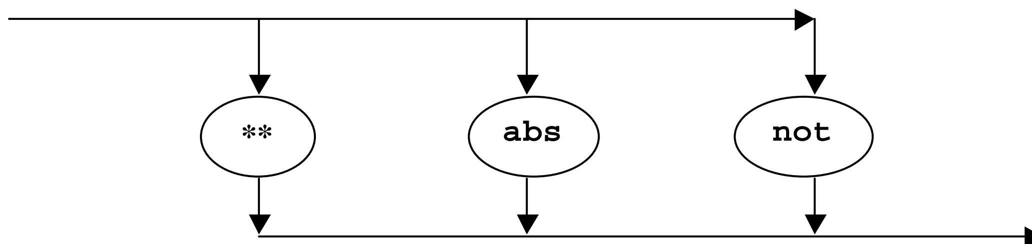
`3.2, ELEMENT, T_ENTIER'LAST` …. sont des facteurs en tant que primaire ;
`not fin_ruban ; abs TAB(2) ; PI**2` sont des applications des D.S.

remarque : `not(fin_ruban)` est l'application de l'opérateur `not` sur une expression parenthésée (mais qui est un primaire) ! donc est bien aussi un facteur comme avec `not fin_ruban` !

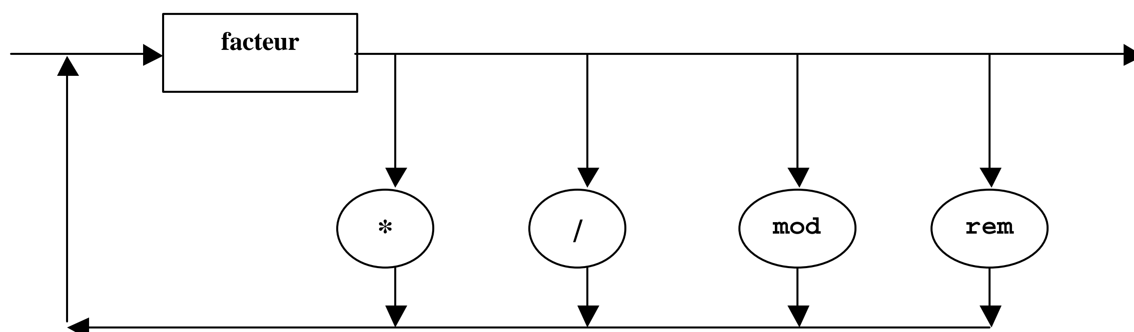
Facteur (D.S.) :



Opérateur prioritaire (D.S.) :



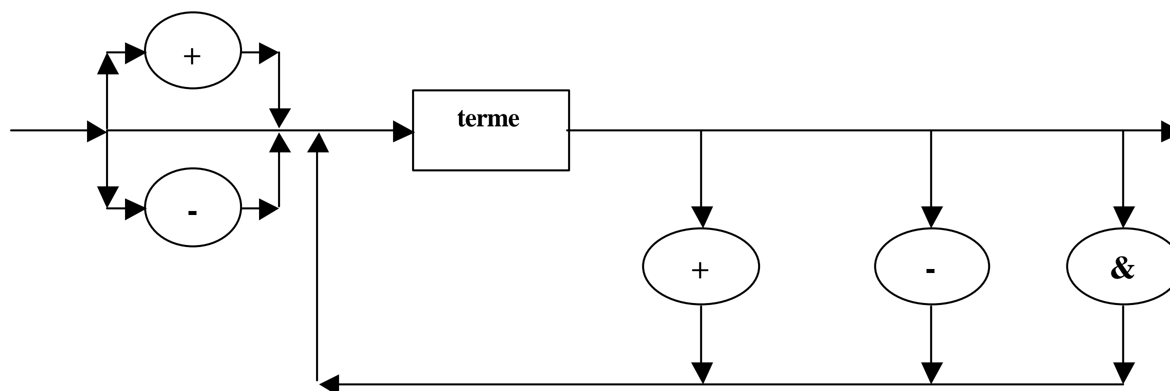
Terme (D.S.) :



On rappelle que les opérateurs ci dessus sont appelés opérateurs multiplicatifs (cours n°1 !). les exemples sont évidents !

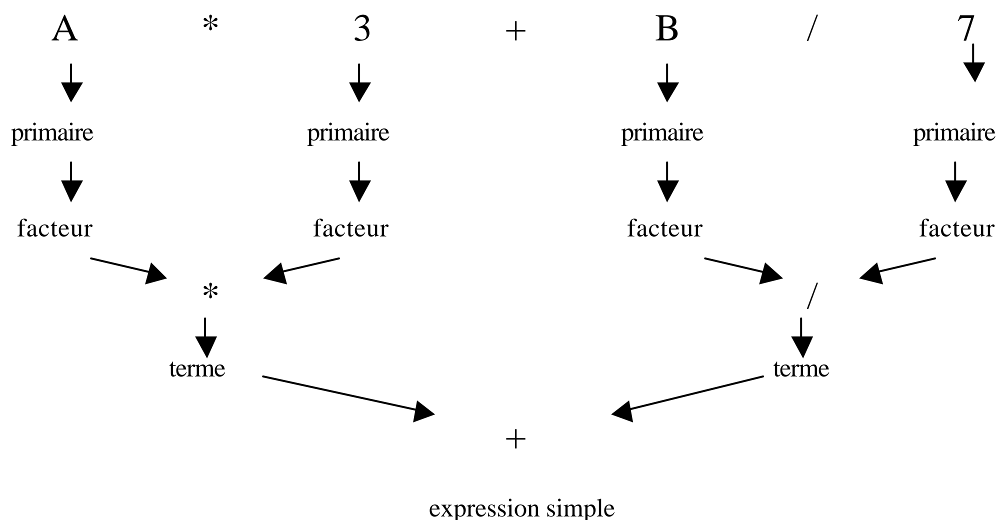
© **Copyright D. Feneuille I.U.T. Aix 2002** (cours n°13 fichier Cours13.DOC)

Expression simple (D.S.) :



Remarque : les deux premiers opérateurs sont dits unaires additifs et les trois autres sont binaires additifs. Bien sûr le dernier est un opérateur agissant, a priori, sur tout type tableau caractères et/ou élément de tableau.

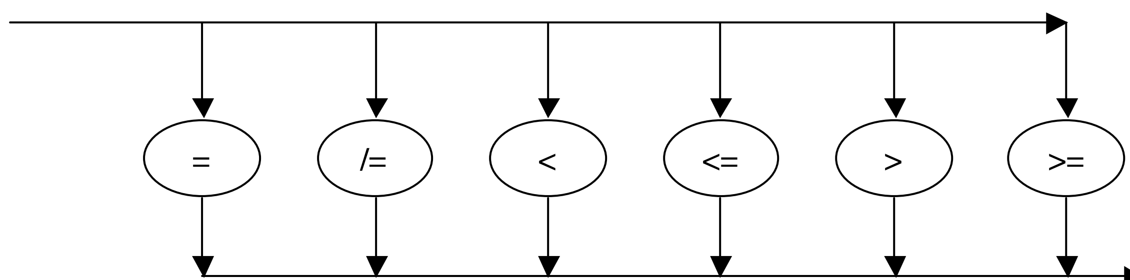
Exemple : A * 3 + B / 7



Relation :

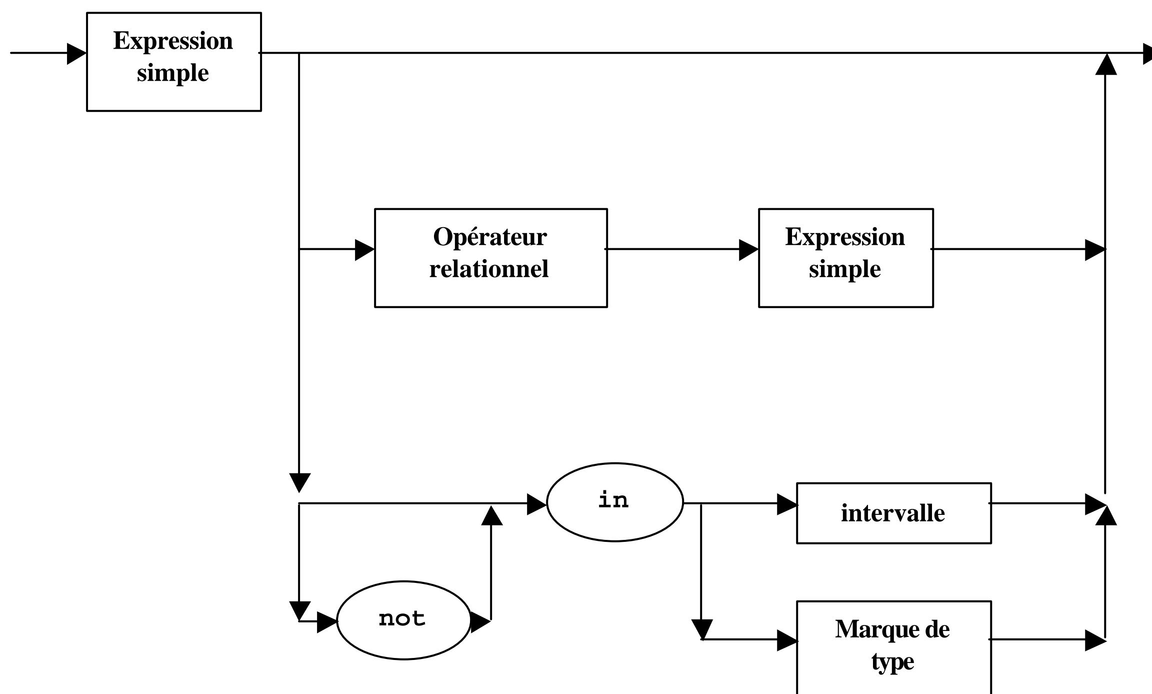
Voyons d'abord les opérateurs relationnels (sans surprise cf. cours n°1) :

Opérateurs relationnels (D.S.) :



ó **Copyright D. Feneuille I.U.T. Aix 2002** (cours n°13 fichier Cours13.DOC)

Relation (D.S.) :



Exemples :

```
IND not in BLANC..ROUGE ;

TEINTE in T_COULEUR

"bonjour" >= MOT(1..5)
```

Expression :

On voit (cf. le D.S. page suivante) qu'une expression est une relation éventuellement constituée d'opérateurs (dits logiques). Ces opérateurs logiques arrivant en dernier dans les définitions sont évidemment les moins prioritaires, puis vient dans l'ordre les opérateurs relationnels (cf. D.S. relation) puis les opérateurs additifs (cf. D.S. expression simple) puis les opérateurs multiplicatifs (cf. D.S. terme) et enfin les opérateurs (dits prioritaires cf. D.S. facteur). Les diagrammes suffisent à expliquer les priorités des opérateurs sans qu'il soit besoin d'en dresser une liste hiérarchique.

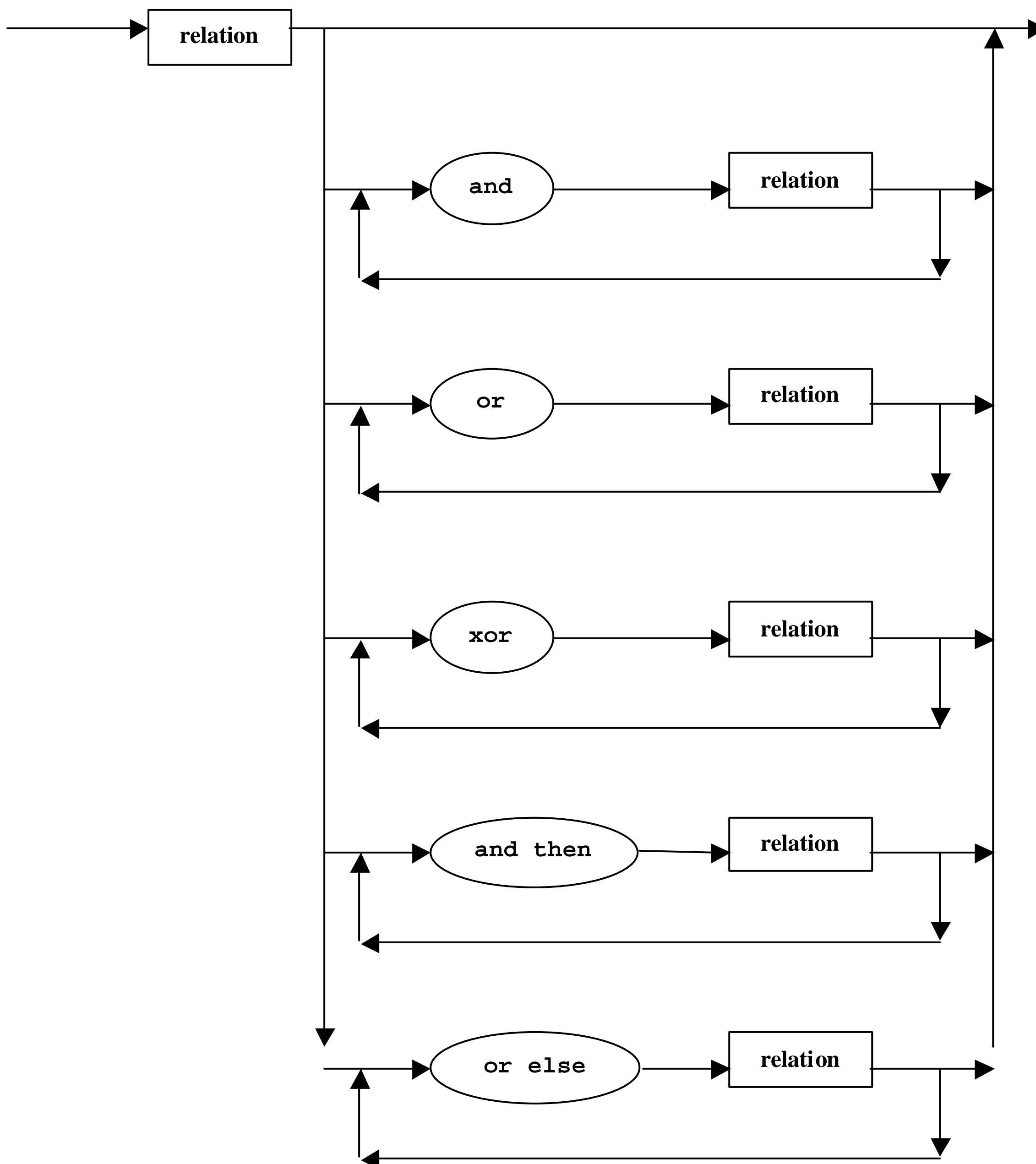
On voit aussi que les opérateurs `and then` et `or else` sont apparentés aux opérateurs `and` et `or`. On rappelle que la différence réside seulement dans l'ordre d'évaluation de leurs opérandes. Dans le cas de `and` et `or` les deux opérandes sont **toujours** évalués tandis qu'avec `and then` ou `or else` l'opérande de droite n'est évalué que si nécessaire (ces deux opérateurs sont appelés opérateurs logiques en raccourci !).

Exemples :

```
(VAL  >= 2) and (RES < 6)
"le résultat est : " & T_ENTIER'IMAGE(RES)
EST_VIDE xor EST_ACTIVE
(1.5 * X**2) - (6.5 * X + 4.7)
```

© Copyright D. Feneuille I.U.T. Aix 2002 (cours n°13 fichier Cours13.DOC)

expression (D.S.) :

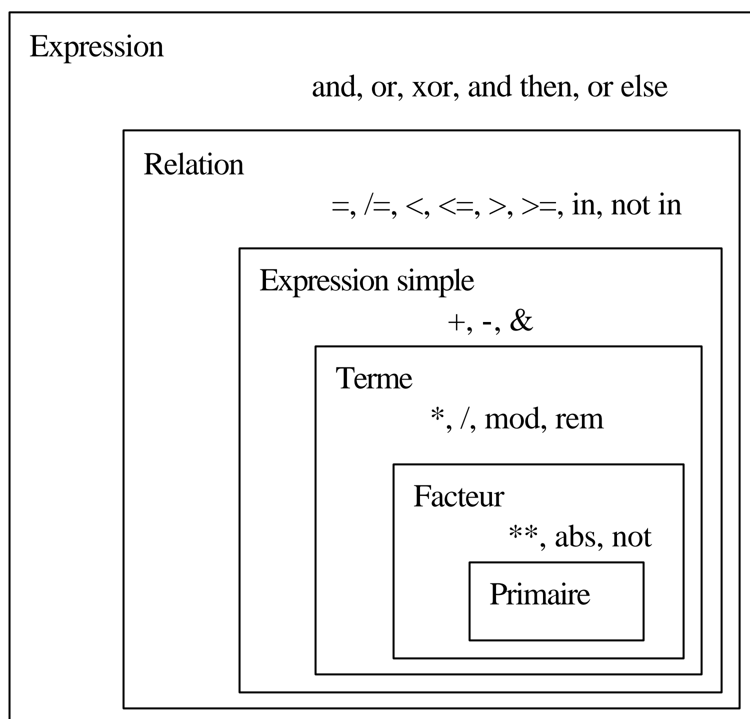


La composition de tous ces diagrammes montre que l'on passe de la définition générale **expression** à la définition particulière d'une entité la composant par emboîtements successifs suivant le principe illustré page suivante.

Remarques et réponses (cf. l'introduction orale au début du cours sous forme de « devinette »)

`A and B and C` est syntaxiquement correct (cf. D.S. ci-dessus)
`A and B or C` est syntaxiquement **incorrect** (cf. D.S. ci-dessus)
`A and (B or C)` est syntaxiquement correct
`(A and B) or C` qui est différent de `A and (B or C)` est également syntaxiquement correct

ó Copyright D. Feneuille I.U.T. Aix 2002 (cours n°13 fichier Cours13.DOC)



Retour sur la notion de type :
Une expression est dite de type entier, réel, booléen, caractère etc. suivant le type du résultat de son évaluation. Ada étant fortement typé les opérateurs ne sont définis que pour certains types. On rappelle aussi que l'opérateur ** n'est défini que pour les puissances entières (ie. deuxième opérande entier).

Exercices :

- Recherchez les D.S. ou les définitions où est apparu le terme d'expression
- `3 ;` qu'est-ce ?
- `D := A+B*(-C) ;` identifiez les différents éléments du membre de droite de cette affectation.
- Soit `(not EXISTE or TROUVE) and JUSTE`. Quelle est la négation de cette proposition.
- `3 + - 4 ;` commentez !
- dans quel ordre s'effectuent les opérations `A + B + C + D ;`
- dans quel ordre s'effectuent les opérations `A * B and C / D ;`
- que pensez-vous de : `INF <= X <= SUP` ?
- que pensez-vous de : `INF <= X and X <= SUP` ?
- comparez les deux expressions suivantes :
  `1.5 * X**2 + 6.5 * X + 4.7 et (1.5 *( X**2)) + (6.5 * X) + 4.7`

Et en B.N.F. ? voici un extrait trouvé dans AdaGide (autrement plus compact !) :

expression ::= relation{and relation} | relation{and then relation} | relation {or relation}
| relation {or else relation} | relation {xor relation}

relation ::= simple_expression [relational_operator simple_expression] | simple_expression [not] in range
| simple_expression [not] in subtype_mark

simple_expression ::= [unary_adding_operator] term {binary_adding_operator term}

term ::= factor {multiplying_operator factor}

factor ::= primary [** primary] | abs primary | not primary

primary ::=
numeric_literal | null | string_literal | aggregate | name | qualified_expression | allocator | (expression)

© **Copyright D. Feneuille I.U.T. Aix 2002** (cours n°13 fichier Cours13.DOC)

Cours 14 Ada FIN ! (2 heures)

Thème : De tout un peu !

Pour terminer ce cours (mais on pourrait y passer encore bien des heures !) nous allons survoler quelques concepts (évoqués éventuellement dans les cours antérieurs) en essayant d'approfondir sans aller au fond (hélas !). J'ai parfois, dans mon cours, emprunté à des petites notes qu'un chantre de Ada diffuse à la communauté Ada. Il s'agit de J.P. Rosen (j'ai déjà fait la pub de son livre) il entretient un site Internet que je vous engage à découvrir (http://perso.wanadoo.fr/adalog/) les notes techniques sont là. Il m'a également autorisé à reprendre la conclusion de son livre (voir à la fin de ce chapitre 14).

Au menu :

1. Des clauses de représentation.
2. Les pragmas (Elaborate notamment).
3. Les annexes Ada du manuel.
4. Les générateurs aléatoires.
5. Interfaçage.
6. Ada et Java.
7. Divers.

1. Quelques clauses de représentation (attention ≈ programmation système) **.**

- **for use** : déjà vu (avec delta pour forcer le pas) mais voici un autre exemple intéressant :

```
type T_Commande is (avant, arrière, arrêt, …) ;
for T_Commande use (0,1,2,4,8, …) ;
```

on a ainsi obligé l'implémentation à prendre (avec 0, 1, 2, 4, 8 etc.) des valeurs formées au plus d'un 1 binaire

- `Size` : permet de fixer par contrat (**for** …. **use** …) la taille d'un objet.
  Par exemple :

```
type T_Octet is mod 256;
for T_Octet'Size use 8 ;
```

Les instances de ce type occuperont désormais 8 bits ! Même si l'implémentation ne faisait pas cela a priori.

- `Small` : permet de fixer par contrat (**for** …. **use** …) la valeur du « Pas » d'un type réel point fixe (déjà vu en cours numériques). Rappel ! Voir notamment le TP sur les Probabilités.

- `Address` : permet de fixer par contrat (**for** …. **use** …) l'adresse d'implémentation d'une variable.

```
Errno : T_Entier ;
for Errno'Address use at...... ;
```

Il faut évidemment bien connaître les adresses mémoire (donc non spécialiste s'abstenir !).

- `at` : permet de fixer par contrat (**for** …. **use** …) un champ de record par rapport à son premier octet d'implémentation.
  Par exemple :

```
type T_Article is record
    Champ_1 : Character ;
    Champ_2 : T_ID ;
    Champ_3 : T_Entier ;
    end record ;
```

```
for T_Article use
record
Champ_1 at 0 range 0..7 ; -- premier octet complet
Champ_2 at 1 range 0..0 ; -- deuxième octet un bit !
Champ_3 at 1 range 4..19 ; -- tiens un trou de 3 bits!
end record ;
```

- `Alignment` : permet de fixer par contrat (**for** …. **use** …) l'alignement sur une frontière d'unités de mémoire d'une variable.

```
for T_Article'Alignment use 2 ;
```

L'article `T_Article` est aligné sur une frontière de double mot.

Exemple :

Voici la déclaration dans le langage C d'un type « équivalent » à un article Ada :

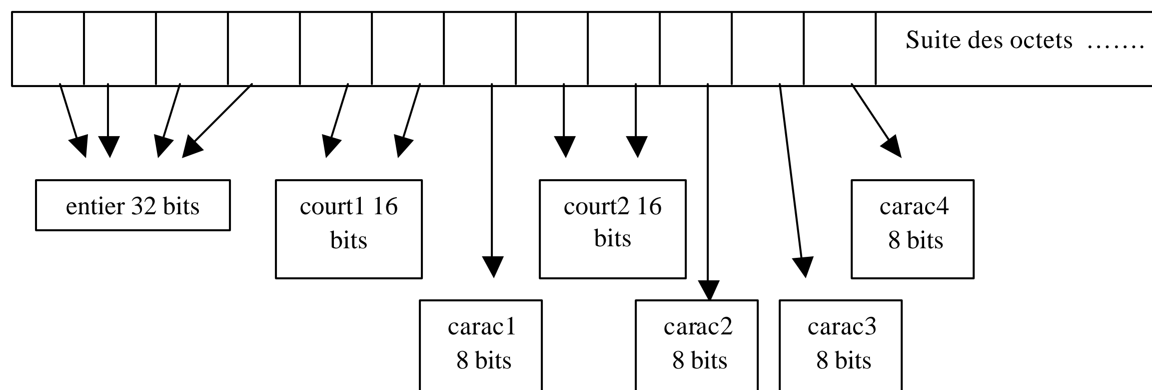
```
struct T_Struct {
     int       entier ;
     short     court1 ;
     char      carac1 ;
     short     court2 ;
     char      carac2, carac3, carac4 }
```

En Ada voici la déclaration « analogue » (je ne dis pas équivalente !) :

```
type T_Rec is record
     Entier : Integer ;
     Court1 : short_Integer ;
     Carac1 : Character ;
     Court2 : Short_Integer ;
     Carac2,
     Carac3,
     Carac4 : Character ;
end record ;
```

En **gras** les mots réservés dans chaque langage.

En C la norme **impose** aux composants de la structure d'être générés de façon contiguë. Soit la suite d'octets :



En Ada rien de cela n'est imposé au compilateur il fait comme il préfère !! Que faire si l'on souhaite à terme interfacer Ada et C ? Il faut imposer à Ada de générer du C ainsi(et dès la fin de la déclaration de `T_Rec`) :

```
for T_REC use record at mod 8

     Entier at 0 range 0..31 ;
     Court1 at 4 range 0..15 ;
     Carac1 at 6 range 0..7 ;
     Court1 at 7 range 0..15 ;
     Carac1 at 9 range 0..7 ;
     Carac2 at 10 range 0..7 ;
     Carac3 at 11 range 0..7 ;

end record ;
```

2. Les pragmas (`Elaborate` notamment).

- `Optimize` (déjà vu avec `Time` et `Space`). Est-ce que c'est efficace sur nos PC avec linux et Gnat ?
- `Pack`. Permet de compacter une implémentation de tableau en général. Voir déclaration des `String` dans le paquetage `STANDARD`.
- `Import` et `Export` (voir le chapitre interfaçage plus loin).
- `Assert` (vu en algorithmique très intéressant !)
- `Elaborate`.

Exemple

Que pensez vous du programme suivant :

```
procedure Problème is
…
function INIT return T_Type ;
….
Val : T_Type := Init ;
…..
     function Init return T_Type is
     begin
      return T_Type'Last ;
     end Init ;
….
begin – Problème
….
end Problème ;
```

Il y aura levée de l'exception Program_Error ! car le corps de Init n'a pas été élaboré.

Supposons maintenant que la fonction `Init` est déclarée dans un paquetage `P_INIT` et définie dans le corps du paquetage. Soit alors une utilisation :

```
with P_INIT ; use P_INIT ;
procedure Problème is
     Val : T_type := Init ;
begin
     …..
end Problème ;
```

Le problème n'est pas forcément résolu !

Pour que l'affectation de INIT à Val ne pose pas de problème (à l'exécution) il faut que le corps du paquetage P_INIT soit connu du système (donc : élaboré). Or **seule l'élaboration de la partie spécification** de P_INIT est garantie par le **with**. Pour être sûr il faut forcer le système à élaborer le **body** avec la ligne supplémentaire :

```
with P_INIT ; use P_INIT ;
pragma Elaborate (P_INIT) ;
procedure Problème is
```

On peut forcer ceci **de façon automatique** dans la **partie spécification** de P_INIT avec le pragma Elaborate_Body. Mais ce n'est pas toujours possible (imaginez un corps de parent utilisant ses propres enfants !). Enfin le pragma Elaborate n'est pas transitif dans une succession de paquetages. Il faut alors utiliser Elaborate_All.

Que faire alors ? On vous recommande de mettre systématiquement Elaborate_Body dans vos spécifications de paquetage. C'est efficace dans 99% des cas. Si le compilateur n'est pas content réglez les problèmes au coup par coup ! Pour en savoir plus voir l'article technique n° 8 de chez Rosen. (et les détails de Elaborate_All)

3. Les annexes Ada du manuel (repérés par des lettres A, B, ….)**.**

A. Précise l'environnement prédéfini. Voir Ada GIDE.
B. Précise les interfaçages avec les langages **normalisés** (C, C++, Fortran, Cobol).
C. Précise les problèmes liés à la programmation système.
D. Précise les problèmes pour le temps réel.
E. Précise les problèmes pour les systèmes distribués et les applications réparties
F. Précise les problèmes des systèmes d'information (gestion).
G. Précise les problèmes des numériques.
H. Précise les problèmes de sûreté de fonctionnement.

Il existe aussi les annexes J, K, L , M, N et P. Cette dernière (P) précise la grammaire de Ada en B.N.F. !!!

4. Les générateurs aléatoires.

Le paquetage Ada.Numerics offre deux « fils » dédiés à la générations de valeurs aléatoires : Ada.Numerics.Float_Random et Ada.Numerics.Discrete_Random.

On peut « deviner » que la valeur aléatoire rendue est **réelle** si on utilise le premier paquetage Float_Random et elle est **entière** avec l'autre paquetage Discrete_Random.

Le deuxième paquetage Discrete_Random est générique sur le type entier en question. Le premier paquetage Float_Random n'est pas générique car la valeur réelle est forcément **contrainte** entre 0.0 et 1.0. Voir dans le polycopié « paquetages » les paquetages respectifs.

La fonction (dans les deux cas) qui rend la valeur aléatoire est identifiée par Random.

Des suites d'appels de Random donne toujours la même suite de valeur **si l'initialisation du générateur** (type Generator) est la **même** (voir la procédure Reset). Vouloir générer la même suite aléatoire pour bizarre qu'elle paraisse peut être utile quand on veut faire des tests « re-jouables ».

On peut mémoriser une valeur du générateur dans une variable compatible (type State) et même convertir en String (Image et Value).

Voir les trois exemples d'utilisation dans le chapitre A.5.2 de AdaGide. J'utilise cette technique pour tirer au hasard l'ordre des 20 questions dans les QCM machine qui vous ont tant fait souffrir.

5. Interfaçage.

La possibilité de pouvoir mêler du code Ada avec un autre **langage normalisé** (à ce jour: C, Fortran, Cobol et depuis peu C++) permet des perspectives intéressantes. Ces possibilités sont définies dans le paquetage `Interfaces` et ses fils.

- `Interfaces.C`
- `Interfaces.C.Strings`
- `Interfaces.C.Pointers`
- `Interfaces.Fortran.`
- `Interfaces.Cobol.`
- `Interfaces.C++` (dans Gnat 13p voir le répertoire examples)

On notera que les « échanges » se font dans les deux sens grâce aux pragmas `Import` et `Export`.

GNAT dans les dernières version s'interface aussi (et depuis peu) avec C++ (à tester!) et même « avec » Java (voir JGNAT). L'IDE AdaGide offre la propriété JGNAT (voir l'icône le plus à droite en haut)à tester aussi

Le compilateur Object Ada d'Aonix s'interface depuis longtemps aussi avec Java (voir plus bas).

L'annexe B est à consulter pour plus de détails sur l'interfaçage. Nous allons détailler un peu la possibilité de mêler du C dans un programme Ada notamment pour accéder à des couches de bas niveau.

Notez que pour communiquer avec un autre langage il faut que Ada manipule des types qui soient « homogènes » avec leur équivalent dans l'autre langage. D'où pour le C les 3 premiers paquetages cités ci dessus (fils de `Interfaces`). Voyons le premier.

Le paquetage `Interfaces.C` (cf. B3) déclare de **nouveaux types Ada** mais possédant une structure **compatible** avec les types équivalents en C [1]. Par exemple : `int, short, long` sont trois déclarations de types entiers Ada « compatibles » C. Le lecteur qui connaît les identificateurs des 3 types primitifs C équivalents constatent que les 3 identificateurs Ada ont été choisis identiques, en effet, ces identificateurs n'existent pas en Ada (Integer, Short_Integer et Long_Integer [2]). C'est une bonne idée « mnémonique » ! Pour les types réels ont trouve `C_float, double, long_double`. Dans cet exemple le premier identificateur ne peut prendre son label C: `float` car cet identificateur existe en Ada il sera donc nommé `C_float`. Les deux autres sont identiques à leur homologue C (en Ada : Long_Float et Long_Long_Float [3]). On notera aussi `char` et `char_array`. Respectivement `char` pour définir des « sortes » de Character et `char_array` pour définir des tableaux de caractères (String) ; `char` n'existant pas en Ada on peut le choisir ; par contre `string` étant un label C (comme précédemment `float`) implique le `char_array`. On notera aussi dans le paquetage les fonctions `To_C` et `To_Ada` (respectivement) qui permettent de convertir des types prédéfinis Ada en leur équivalent Ada compatibles C ou les types Ada compatibles C en types prédéfinis Ada (respectivement).

Exemple : `function To_C (Item : in Character) return char;`
Permet de convertir un caractère Ada en son équivalent Ada de type `char` pouvant être « offert » à un module C. Ces identificateurs `To_C` et `To_Ada` sont plusieurs fois surchargés il faut les étudier en détail.

Un exemple d'application : (simple pour fixer les idées) « Interfacer une commande système en Ada » :

Problème : vous voulez faire exécuter, dans un programme Ada, une commande système par exemple clear ou cls ou ls ou ls –l (là encore pour fixer les idées)

Dans le langage C, il existe une instruction (notée `system`) qui permet de lancer une telle commande. Il suffit alors d'interfacer, en C, la fonction `system` avec Ada. Ainsi :

---

[1] On rappelle que contrairement à C en Ada la norme n'impose pas une structure définie aux types numériques par exemple.
[2] Pour fixer les idées (voir la remarque ci dessus).
[3] Toujours pour fixer les idées comme précédemment.

```
with Ada.Interfaces.C ;
-- permet l'association d'éléments Ada et des éléments compatibles C
……
use Ada.Interfaces.C ;
-- pour éviter de préfixer !

procedure C_System (Chaine : in Char_Array);
-- profil Ada de C_System donc déclaration de C_System
-- cette procédure sera « l'équivalente » de l'instruction system de C

pragma Import (C, C_System, "system") ;
-- réalisation de C_System avec la procédure C notée system
……
begin
….•
     C_System(To_C(chaîne Ada illustrant la commande)) ;

     -- utilisation de C_System avec transformation de la chaîne Ada
     -- en son équivalent en C grâce à la fonction de conversion To_C
……
```

Revoyons sur cet exemple, facilement généralisable, les trois phases (déclaration, réalisation et utilisation) de la procédure Ada qui est écrite en C.

Déclaration (spécifications) : elle se nomme `C_System` (on aurait pu la nommer autrement (`système` ou même aussi `system` ! son vrai nom en C). En paramètre il faut la chaîne de caractères représentant la commande à lancer mais attention cette chaîne de caractères est une chaîne Ada compatible C donc de type `Char_Array`.

Réalisation : il n'est pas question d'écrire le codage mais simplement d'expliquer au compilateur (en fait à l'éditeur de liens !) de prendre en lieu et place de la procédure Ada `C_System` la procédure C `system`. Notez le pragma avec son nom `Import` puis en paramètre le nom du langage d'interfaçage, le nom de la procédure Ada à réaliser puis en troisème paramètre et entre guillemets le nom de la procédure de substitution. Si au lieu de prendre la procédure connue `system` on voulait utiliser une autre procédure écrite en C (mais pour un autre problème) c'est la même écriture ; il faut simplement s'assurer que l'éditeur de liens trouve le fichier .ali du code C compilé.

Utilisation : dans le programme Ada on utilise en nommant la procédure `C_System` avec son vrai paramètre effectif entre parenthèses et c'est la chaîne de caractère illustrant la commande tels que :

```
C_System(To_C( ls -l )) ;
-- affiche les répertoires linux !

C_System(To_C( clear )) ;
-- met l'écran à blanc en linux

C_System(To_C( cls )) ;
-- met l'écran à blanc en Windows
```

Voir dans le répertoire fendan/util la procédure envoi (fichiers envoi.adb) qui permet d'envoyer un mail dans un programme Ada.

Notez bien que `C_System( clear )` n'est pas correct car `clear` est un littéral chaîne Ada et non de l'Ada compatible C ! Il faut convertir avec la fonction `To_C`.

Il existe aussi un **utilitaire intéressant et gratuit** qui transforme en Ada un séquence codée en C. Il s'agit du translator C2Ada. Il n'est pas efficace à 100 % mais le reste à finir (à la main) n'est pas volumineux (dixit J.P. Rosen). Voir à l'adresse: http://www.inmet.com/~mg/c2ada/c2ada.html pour en savoir plus.

Notez que maintenant on peut dire que : « si cela existe pour C alors cela existe pour Ada » !

6. Ada et Java.

A titre d'anecdote on peut signaler qu'à défaut d'interfacer vraiment Java certains compilateurs (Object Ada ou Intermetrics par exemple) permettent d'écrire en Ada des séquences de Java. Comment cela est-il possible ?

Java est un langage (assez ressemblant à du C++ épuré). Son intérêt est de produire du code (JBC) pour une machine virtuelle et portable (à exécuter sur la machine cible). Par exemple sur Internet c'est la machine, de celui qui lit la page Web ou l'applet, qui exécute le code envoyé par le site sur lequel on s'est connecté. C'est assez intéressant car spectaculaire (informatique répartie simple en IntraNet) mais **cela peut être très lent**. Pour en savoir plus (sur Java) on pourra consulter le CDRom pour commencer où j'ai simplement fait un survol de la première version de Java le 1.1. On trouvera aussi dans les « pointeurs » Internet quelques sites où de bons cours sont proposés notamment chez Durif : www.lifl.fr/~durif/ .

Le compilateur Ada permettant les applets génère tout simplement du JBC ! Rosen appelle les applets générés par Ada des Adapplets ! Voir http://www.appletmagic.com : the game of Mancala (démo et sources !).

Mais on l'a rappelé plus haut Gnat propose maintenant JGNAT pour faire la même chose. Pour le moment la version est en phase de test. A revoir. C'est aussi sur le CDRom pour les fanatiques. Le site www.gnat.com actualise les versions. A consulter.

7. Et encore des petites choses.

- `Unchecked_` : on a déjà vu `Unchecked_Deallocation` (danger !) et `Unchecked_Access`. Voici `Unchecked_Conversion`. On peut forcer Ada à « détyper » un objet par exemple un caractère en une valeur numérique contrainte entre 0 et 255 (ah le langage C, mon cher, il permet cela depuis longtemps ! Oui mais sans être obligé de l'annoncer et c'est bien là le danger !).

```
fonction Char_To_Byte is new Unchecked_Conversion (Character, T_Octet) ;
    ............
VAL := 2* Char_To_Byte(Carcou) ;
```

- `Command_Line`. Ce paquetage « fils » de Ada permet de récupérer la ligne de commande de lancement du programme (comme avec $1, $2 etc. dans les scripts shell pour fixer les idées). Toutes les occurrences (sous forme de chaînes de caractères) sont récupérables. Depuis la commande elle même, identifiée `Command_Name`, ainsi que les « arguments » s'ils existent. La fonction `Argument_Count` rend le nombre d'arguments (0 à ....) et `Argument(IND)` rend la chaîne correspondant à l'argument numéro `IND`. Voir le paquetage `Ada.Command_Line` dans le polycopié « paquetages ». C'est très simple (cela existe en Pascal, en C, etc.) mais c'est bien utile. Attention nostalgique de Pascal, notez que `Argument(0)` n'est pas correct ! si vous souhaitez récupérer le nom de la commande elle même en chaîne de caractères c'est `Command_Name` comme indiqué plus haut.

Exemple :
Si on lance par exemple la commande suivante `./essai param1 param2`

Le programme `essai` a deux paramètres dans la commande et l'on peut obtenir :
`Command_Name = "essai"`, `Argument_Count = 2`, `Argument(1) = "param1"`, `Argument(2) = "param2"`.

- **Compilateurs**. Les plus connus en France sont :

  - GNAT (version commerciale chez Ada Core Technologies) « sans commentaire ».
  - Object Ada (société Aonix) décliné en plusieurs versions. Enorme ! permet des GUI et même Java !
  - Apex (société Rational Software).
  - Intermetrics
  - etc.

- Surcharge de =. On peut surcharger cet opérateur (et même le définir pour les types limités). A noter que la valeur de retour peut être non booléenne (puisque l'on parle de surcharge!) mais danger d'interprétation.

- Des fichiers à consulter.

Sans rappeler tous les fichiers .doc ou .ads et .adb que vous deviez éditer pour parfaire votre culture Ada je cite ici ceux qui auraient peut-être été oubliés :

Pragmas.doc
Attribut2.doc
Attribut3.doc
Article1.doc
Article2.doc
Chaine.doc
glossaire.doc
etc. voir sur le CDRom le répertoire compléments (à compléter!?)

- TASH : est une sur-couche de Tcl/tk qui est lui même un langage de script permettant de réaliser des interfaces graphiques. On pourra voir à l'adresse http://tash.calspan.com le fichier README.htm

GtkAda.

GtkAda est un ensemble d'outils (toolkit) graphiques permettant de réaliser en Ada des interfaces utilisateurs (G.U.I.) de qualité.
GtkAda s'appuie sur Gtk+ (on parle de portage), Gtk+ est lui même un toolkit graphique utilisant un ensemble de bibliothèques qui sont développées en C. De façon transitive on peut donc dire que GtkAda une sur-couche de C pour réaliser cet objectif.
Le portage de Gtk+ existe aussi pour de très nombreux langages (Eiffel, C++, JavaScript, Pascal, Perl, Python, etc.). Nous l'avons utilisé pour donner un bon look aux TP d'algorithmique. Sans oublier les horribles QCM qui vous ont plus ou moins contraints à réviser votre cours.
Pour le découvrir visitez http://gtkada.eu.org notamment le fichier gtkada_ug.htm pour en savoir plus on peut aussi voir mon essai de traduction de ce fichier sur le CDRom.
Si vous préférez la langue de Molière vous trouverez une sérieuse documentation de 65 pages à l'adresse suivante : http://www.lifl.fr/~durif/ . Oui mon collègue lillois Durif le même qui a écrit un bon cours de Java cité précédemment.

Héritage multiple.

L'héritage multiple (au sens P.O.O.) est la propriété pour une classe (en Ada un type tagged rappelons le) de dériver de plusieurs ancêtres. En Ada il n'y a pas de structure syntaxique pour réaliser cette construction. Cette notion d'héritage multiple est **très séduisante mais pas sans problèmes**. En effet étendre une classe, à partir de deux classes par exemple, et hériter de toutes les familles c'est bien mais que se passe-t-il quand les deux classes héritées ont, elles mêmes, le même ancêtre ? Ce problème « excite » beaucoup les informaticiens.

En Ada peut-on, pour autant, faire de l'héritage multiple ou quelque chose qui lui ressemble ? Evidemment la réponse est oui (mais pas de façon satisfaisante !). Il « suffit » quand on veut « hériter » de deux types (dans deux paquetages donc) d'en privilégier un et de faire sur le type sous-jacent la dérivation vraie (**new** … **with record** …) et « d'hériter » de l'autre type en s'appuyant sur son paquetage uniquement avec un **with** (ou de faire un fils) ou encore grâce à la généricité. Ainsi on récupère les fonctionnalités que l'on peut filtrer éventuellement. A voir si le temps le permet mais ce n'est pas évident après un trimestre d'informatique.

Les dix mythes ou idées fausses sur Ada[4]

Florilège de « conneries » qui **courent** sur Ada

Ada est un langage trop complexe.

Faux ! Ceux qui disent cela se sont contenté de survoler le manuel de référence qui évidemment n'est pas un document minimum puisqu'il spécifie complètement le langage. Si Ada était si complexe pourquoi aurait-il tant de succès comme langage idéal pour enseigner l'informatique (mais la vraie !).

Ada coûte trop cher.

Faux ! Les compilateurs Ada professionnels incluent automatiquement des outils qui sont d'habitude fournis en complément pour d'autres langages concurrents et globalement la différence n'existe pas. Et évidemment le Gnat propose une version gratuite qui n'en est pas moins remarquable.

Ada n'est-il pas un langage pour les militaires ?

Certes le D.O.D. est l'instigateur du langage et a soutenu sa généralisation. Mais aujourd'hui Ada est autant apprécié par des universitaires et est aussi utilisé dans de nombreuses applications commerciales.

Ada n'est pas utile pour développer de petites applications.

Pourquoi pas. Certes Ada est incontournable pour des applications de plus de centaines de milliers de lignes, mais il permet d'écrire aussi bien de petites choses plus modestes. Qui peut le plus, peut le moins !

Ada ne permet pas de souplesse dans la programmation.

Faux ! La souplesse existe mais elle n'est pas implicite et elle doit être annoncée à l'avance car le compilateur est très rigoureux. Rappelons aussi que Ada s'interface avec de très nombreux autres langages.

Ada n'est pas répandu.

Si l'on se réfère uniquement aux offres d'emploi alors on peut dire cela. Est-ce une raison pour le rejeter. Diriez vous que TF1 est la meilleure chaîne de télévision au prétexte qu'elle est la plus regardée ?

Ada n'est pas pour les programmeurs chevronnés.

Oh certes les bidouilleurs, les fanas de la programmation système, de l'assembleur, du développement sans lendemain, du logiciel jetable ne trouvent pas leur compte dans la rigueur de Ada. Tant mieux !

Ada est trop verbeux.

Ceux qui disent cela n'ont jamais dû programmer en Cobol ! Ah certes Ada n'a pas la compacité du C. Mais la lisibilité sous jacente de cette prétendue verbosité entraîne une bien meilleure lisibilité qui documente mieux l'application. On peut toujours trouver des contre exemples pour convaincre en voici un :
`Vect(5..7):= Vect(10..12)` contre `memcpy(vect+5,vect+10,3*sizeof(*vect))`

Ada est trop lent et génère du code volumineux.

Faux ! le Gnat utilise les mêmes environnements (gcc par exemple) que les autres langages du GNU. Il existe aussi des directives pour obtenir des optimisations souhaitées (revoyez l'option –s par exemple).

Ada n'est pas orienté objet.

Totalement faux ! Une lecture trop hâtive d'un code Ada à la recherche du constructeur syntaxique class peut laisser supposer cela ! Mais Ada 95, au contraire, a été le premier langage normalisé permettant les objets ! Et Ada propose aussi exceptions, généricité et tâches depuis des lustres.

[4] D'après un article trouvé chez : http://home.hiwaay.net/~crispen/ada/top_10.html

Ada « c'est la fin ! » (en guise de conclusion)

J'ai longtemps galéré pour composer une conclusion digne de ce langage qui me ravit au delà du raisonnable comme vous avez pu le constater dans mes interventions parfois partisanes. En lisant (et en savourant) dans le livre de J.P. Rosen[5] « Méthodes de génie logiciel avec Ada 95 » la conclusion qu'il a su en tirer, je renonce à faire mieux! Avec son consentement, vous trouverez ci dessous cet excellent papier : je me suis juste permis d'ajouter de petites choses (dans les notes de bas de page), de mettre en page différemment et de surligner des concepts importants. Bonne lecture.

Arrivé à ce point, le lecteur que nous avons pu convaincre de l'intérêt d'Ada se posera certainement la question : **pourquoi Ada n'est-il pas plus répandu** ?

On pensait, au début (vers 1980), que Ada remplacerait rapidement les langages alors principalement utilisés : Fortran, Pascal et, dans une moindre mesure, Cobol. Or si le langage Ada s'est bien introduit dans les domaines sensibles (aéronautique, aviation, domaine militaire, nucléaire, contrôle de processus), sa diffusion est restée modeste dans les domaines de la programmation dite traditionnelle : scientifique, gestion, programmation système, alors qu'on assistait à une montée en force du langage C, et plus récemment (depuis 1990) de C++.

Pourtant, toutes les mesures effectuées sur des projets réels ont montré que les bénéfices que l'on pouvait espérer d'Ada étaient obtenus ou dépassés : meilleure qualité de la conception, réutilisation, augmentation de la productivité des programmeurs, infléchissement de la courbe des coûts en fonction de la taille des logiciels, effondrement du taux d'erreurs résiduelles et des coûts d'intégration, efficacité du code produit ; et ceci, quel que soit le domaine d'application concerné : on trouvera un exemple dans une étude complète sur l'impact de l'utilisation de Ada en gestion[6].

Mais le choix d'un langage de programmation fait intervenir de nombreux éléments, qui touchent plus à la psychologie qu'à l'économie.

Tout d'abord, l'expansion de C est liée au développement du système UNIX. Il est de tradition de fournir gratuitement le compilateur C avec les outils standard d'UNIX[7]. Pourquoi alors acquérir un autre langage ? D'autant plus que C a pour lui une extrême simplicité (d'aucuns disent même pauvreté) de concepts. Remarquons au passage que ces arguments ne s'appliquent plus à C++ : les compilateurs sont payants[8], et le langage est devenu bien plus compliqué !

Historiquement, UNIX a été développé par un petit nombre d'individus, sans souci de politique d'ouverture commerciale. Les interfaces ont donc été pensées uniquement en fonction des particularités du C ; c'est ainsi qu'une fonction peut retourner soit un pointeur, soit la valeur False (c'est à dire en fait zéro, qui est également le pointeur nul !). De telles hérésies de typage sont incompatibles avec la plupart des langages évolués, mais fréquentes avec C. Il en résulte des difficultés à faire des interfaces abstraites, propres, pour beaucoup de fonctionnalités UNIX, ce qui accentue le retard, en particulier au niveau de la standardisation, des autres langages par rapport à C. Et puis la « sagesse populaire » affirme que l'on doit programmer en C sous UNIX, sans plus d'explication. Bien sûr, il existe de nombreux composants et de nombreuses interfaces adaptés à Ada. Mais les vendeurs font rarement un effort commercial conséquent pour leur promotion, et il faut les acheter en plus[9], alors que beaucoup de bibliothèques C sont fournies avec le système. Là encore, l'effort initial est plus important avec Ada, même s'il est aisé de prouver que ce surcoût au départ est largement amorti sur le long terme.

En dehors de ces considérations pratiques, l'attirance du programmeur moyen pour C, et la peur instinctive qu'il éprouve vis à vis de Ada, plongent leurs racines bien plus profondément. Beaucoup de programmeurs encore actuellement en fonction ont été formés à une époque où l'assembleur était roi. Nous avons connu des

---

[5] Largement évoqué de nombreuses fois dans mon cours !
[6] Thèse de Riadh Lebib (94) «Incidences de la mise en œuvre des concepts de génie logiciel à travers l'utilisation du langage de programmation Ada sur la productivité en informatique de gestion ».
[7] Ceci tend à disparaître... ce qui fait une raison de plus d'utiliser Ada !
[8] Sauf évidemment avec Gnat !
[9] Sauf évidemment avec Gnat ! (Bis !)

centres de calcul où les « nobles » (ceux qui étaient capables d'écrire tous leurs programmes en langage machine) regardaient de haut les novices qui n'étaient bons qu'à programmer en Fortran... Avec le temps, la programmation en assembleur tend à disparaître, car les problèmes de fiabilité, de maintenabilité et de portabilité sont vraiment devenus trop importants. Que sont devenus ces programmeurs [10] ? Eh bien ils ont fait et font sûrement encore du C. Ils ont trouvé un langage qui n'est en fait, selon la définition de ses auteurs, qu'un assembleur portable [11]. Comme nous l'avons vu, il permet de faire (presque) tout ce que l'on pouvait faire en langage machine et n'implique aucune remise en cause ni des méthodes ni de la façon de programmer. Inversement, Ada a été spécifiquement conçu pour obliger les programmeurs à modifier leurs habitudes ! En particulier, la programmation Ada s'accompagne d'une description plus abstraite des traitements. Outre que ceci nécessite un effort de réflexion plus important, le programmeur tend à perdre le contact direct avec la machine. Il doit faire confiance au compilateur pour la génération de code efficace, et ne doit plus se préoccuper du parcours exact du programme. Il n'est guère étonnant que l'inertie naturelle ne rende pas Ada très attrayant a priori au programmeur qui n'a pas saisi (car il n'a jamais essayé) les bénéfices d'une approche de plus haut niveau. Mais bien sûr, l'attirance naturelle du programmeur ne saurait être un critère pour un choix technologique dont dépendra tout le projet ! Comme il a été remarqué :

C++ ressemble à un énorme gâteau à la crème, ruisselant de sucreries ; Ada ressemble plus à un poisson poché /pommes vapeur. Les questions intéressantes sont :
1) quel est le meilleur pour votre santé
2) qu'est ce que les gens ont tendance à choisir spontanément ?

En outre, les compilateurs Ada ont parfois mauvaise réputation [12]. Les premiers n'étaient disponibles que sur un petit nombre de machines, et peu efficaces, quand ils n'étaient pas franchement « buggés ». Les premières années, l'effort des fabricants s'est plus porté sur la conformité à la norme, vérifiée par l'incontournable suite de validation, que sur l'efficacité pour laquelle il n'y avait aucune obligation légale. Des compilateurs sont maintenant disponibles sur quasiment toutes les machines du marché, et des mesures objectives ont montré qu'ils étaient au moins aussi performants et fiables que les compilateurs C. Il n'en reste pas moins que le langage continue à traîner quelques vieilles « casseroles », d'autant qu'il est difficile de supprimer de la littérature les comptes rendus des premières expériences. C'est ainsi que Burns (en 85) signalait qu'un rendez vous prend 100 ms sur un Vax, ce qui rend le mécanisme inapte au temps réel serré. De nos jours, un rendez vous prend moins de 100 µs (même qu'à 4 µs sur certains systèmes spécialisés), mais le livre de Burns (excellent au demeurant) n'a été réactualisé que fort tardivement.

Un langage ne peut se répandre que s'il est enseigné.

Force est de reconnaître qu'Ada n'a pas encore réussi à séduire tous les universitaires. Au début (1985) le prix des compilateurs, justifié en milieu industriel par le gain de productivité apporté, a été un obstacle certain. C'est ce qui a conduit le DOD à financer le GNAT, pour mettre à la disposition de tous un compilateur Ada 95 gratuit. Ensuite, chercheurs et enseignants ont généralement des contraintes différentes des industriels : les logiciels sont rarement maintenus, mais doivent souvent être développés très rapidement. Même si leurs cours d’informatique prônent le génie logiciel et l'importance de favoriser la maintenance au détriment de la facilité de développement, les universitaires appliquent rarement ces bons principes à eux mêmes [13] ... Inversement, les langages orientés objet sont à la mode et semblent offrir des facilités supplémentaires (mais cet argument n'a plus lieu d'être avec Ada 95). Enfin, le fait que le DOD ait financé le développement de Ada lui a créé une certaine antipathie dans les milieux universitaires, traditionnellement antimilitaristes.

Paradoxalement, les promoteurs du langage ont pu lui faire du tort par excès de purisme en exigeant de faire du 100% Ada.

Le langage a tout prévu pour permettre d'écrire des modules en d'autres langages (C, Fortran, Cobol et même C++ et Java) si c'était plus commode ; il autorise même l'écriture de modules en assembleur. Si ces éléments sont alors non portables, ce n'est pas trop grave dans la mesure où le mécanisme d'empaquetage

---

[10] N'oublions pas que les programmeurs qui étaient débutants à l'époque héroïque (début des années 60) n'ont peut-être pas tous encore atteint l'âge de la retraite ! Nous sommes en l’an 2000

[11] Selon un bon mot de M. Gauthier « Le langage C est, avec Lisp, le plus intransportable des assembleurs transportables, classes dont ils sont, d’ailleurs, les seuls représentants ! ». Je cite de mémoire.

[12] Enfin c’était vrai pour Ada83.

[13] Pardonnez moi, « mes frères », d’avoir ouvert la boîte de Pandore !

garantit que ces non portabilités sont répertoriées, aisément identifiées, et sans effet sur les utilisateurs des paquetages. Il n'y a donc pas de honte à écrire un corps de module en assembleur si, par exemple, il n'y a pas d'autre moyen d'obtenir les performances requises. En exigeant le « tout Ada » au delà du raisonnable, on risque de se heurter à des problèmes tout à fait localisés, mais dont la publicité risque de discréditer le langage tout entier.

Enfin, il est remarquable de constater que les plus virulents adversaires d'Ada... ne l'ont jamais vraiment pratiqué !

Peut être ont ils juste fait quelques essais, sans formation appropriée, en traduisant « littéralement » des bouts de code d'autres langages avec lesquels ils étaient familiers. Au bout de quelques erreurs de compilation, ils ont décidé que le langage était difficile, et n'ont pas poursuivi.

Inversement, ce n'est qu'à l'usage que l'on en apprécie tous les avantages : typage fort, exceptions, parallélisme, paquetages hiérarchiques, etc. Il est certain que pour tirer tout le bénéfice de Ada, il ne suffit pas d'apprendre la syntaxe ; il faut assimiler l'état d'esprit, c'est à dire «la façon de faire » qui va avec. Il faut admettre qu'un langage comme Ada joue un rôle différent dans le processus du développement du logiciel et que, comme on l'a vu, le choix du langage de codage n'est pas neutre : précisément parce qu'il peut être beaucoup plus qu'un simple outil de codage. Comme le remarquait G. Booch en 1991 :

Donnez une perceuse électrique à un charpentier qui n'a jamais entendu parler de l'électricité, et il l'utilisera comme marteau. Il tordra des clous et se tapera sur les doigts, car une perceuse fait un très mauvais marteau.

Si les mauvaises nouvelles courent vite, il est plus difficile de répandre les bonnes.

Ada n'a pas toujours su « faire sa pub ». Qui, en dehors des convaincus, sait qu'Ada a été utilisé avec succès dans de nombreux programmes de gestion, dont un projet de 2,5 millions de lignes de code qui a pu être développé avec la moitié du coût initialement prévu (voir l'article STANFINS-R dit «article2» sur le CDRom) ? Qu'un logiciel qui n'arrivait pas à tenir ses contraintes temps réel a été réécrit en Ada avec des performances satisfaisantes (voir l'article dit «article1» sur le CDRom) ? Qu'une entreprise gère plusieurs projets totalisant un million de lignes de code en n'employant qu'une seule personne à la maintenance (article de H. Pidault « Graphics Development en Ada » dans Ada Europe en 1993) ?

Soyons honnête cependant : les utilisateurs d'Ada 83 ont pu rencontrer des difficultés objectives, même si la plupart ont appris à « vivre avec ». On notera en particulier le manque de support des méthodes par classification, la prolifération des tâches pour de simples synchronisations, de trop nombreuses re-compilations dans les grands projets, des difficultés d'interfaçage avec d'autres langages ou des bibliothèques... et le manque de fiabilité des premiers compilateurs et des environnements de programmation.

Ces problèmes ont disparu avec le temps, ou ont été résolus par Ada 95 et peuvent être maintenant considérés comme résolus.

Enfin (en fin !).

Lorsqu'une étude objective est menée, prenant en compte les principes du génie logiciel, les coûts des développements informatiques pris sur l'ensemble du cycle de vie, et les contraintes de fiabilité et de sécurité, elle aboutit toujours à la même conclusion : Ada est le seul langage qui offre le support nécessaire aux méthodologies modernes de développement. Le problème est que les facteurs objectifs ne sont souvent pas suffisants pour convaincre les utilisateurs. Le bouche à oreille, la rumeur, de mauvais résultats dus à des essais sommaires sont souvent plus crus que les études formelles. Cette attitude se résume en la formule :

« Bien sûr, l'étude a conclu qu'il fallait utiliser Ada... mais mon voisin de bureau travaille en C ».

Il reste encore du chemin à parcourir pour obtenir une approche industrielle du développement logiciel !

Ada c'est plus fort que tout ! Ah !

I.U.T. Aix

Département génie informatique

Génie logiciel

1er trimestre 2001/2002

Le test et la testabilité

Polycopié étudiants

Patrice Micouin

Nom étudiant :

TABLE DES MATIERES

1. GENERALITES 5

1.1. QU'EST-CE QUE LE TEST ? 5
1.2. TESTER N'EST PAS DEBUGGER 5
1.3. QUI TESTE ? 5
1.4. QU'EST-CE QU'UNE ERREUR ? 5

2. LE TEST D'UN SOUS PROGRAMME 6

2.1. ENVIRONNEMENT DE TEST 6
2.2. TYPES DE TEST 8
2.2.1. Test exhaustif 8
2.2.2. Objectifs de la sélection des jeux de test. 8
2.2.3. Test Statistique. 9
2.2.4. Typologie classique des tests. 9
2.2.4.1. Le Test "Boîte Noire" 9
2.2.4.2. Le test "Boîte Claire" 10
2.3. - TECHNIQUES DU TEST STRUCTUREL (BOÎTE CLAIRE) 11
2.3.1. Le test structurel des programmes séquentiels 11
2.3.1.1. Les chemins de contrôle 11
2.3.1.2. Couverture d'instructions 12
2.3.1.3. Couverture de branches (décisions) 12
2.3.1.4. Couverture de conditions 13
2.3.1.5. Couverture de conditions/décisions 13
2.3.1.6. Itérations et récursivité : Quantités de contrôle 13
2.3.2. Le test structurel des programmes parallèles 14
2.3.3. La mesure des temps d'exécution 15
2.4. TECHNIQUES DU TEST FONCTIONNEL (BOITE NOIRE) 16
2.4.1. La technique du partitionnement 16
2.4.2. Le test aux limites 17
2.4.3. Les arbres causes/conséquences 18
2.4.4. La devinette 18
2.4.5. Le test statistique 19
2.4.5.1. Criticité d'un logiciel 19
2.4.5.2. Modèle d'usage d'un logiciel 19
2.5. L'ARRET DES TESTS 20
2.5.1. Arrêt des tests structurels 20
2.5.2. Arrêt des tests fonctionnels 20

3. LE TEST DES MODULES 21

3.1.1. Le test d'un objet 21
3.1.1.1. Comportement d'un objet 21
3.1.1.2. Scénarios de test 23
3.1.2. Le test d'un générique 25
3.1.3. Le test d'une classe 25

4. LE TEST DES GRANDS PROGRAMMES 26

4.1. LES STRATEGIES DE TEST 26
4.1.1. Le test descendant (top-down) 26
4.1.2. Le test ascendant (bottom-up) 26
4.2. LE TEST INCREMENTAL 27
4.2.1. Les tests unitaires 27
4.2.2. Les tests d'intégration 27
4.2.3. Les tests de validation 28

5. LA TESTABILITE ... **29**
5.1. QU'EST-CE QUE LA TESTABILITE ... 29
5.2. TESTABILITE & COMPLEXITE ... 29
5.3. TESTABILITE & SPECIFICATION ... 29
5.4. TESTABILITE & CONCEPTION ... 30
5.5. TESTABILITE & CODAGE ... 30
5.5.1. Les critères de complexité ... 30
5.5.1.1. La taille ... 30
5.5.1.2. Le nombre cyclomatique ... 31
5.5.2. Les conditions de passage en phase de test ... 31
6. OUTILS DE TEST ... **32**
6.1. OUTILS D'ANALYSE STATIQUE ... 32
6.2. OUTILS D'ANALYSE DYNAMIQUE ET DE COUVERTURE ... 32
6.3. SONDES ET EMULATEURS ... 32
6.4. BANCS DE TEST ... 32
7. GUIDE DU TEST BOITE CLAIRE ... **33**
7.1. EXERCICES ... 34
7.1.1. Exercice A : Analyse syntaxique d'un identificateur ... 34
7.1.2. Exercice B : Un algorithme d'insertion dans un arbre binaire ... 34
7.1.3. Exercice C : Le Calcul d'une intégrale par la méthode des trapèzes ... 36
7.1.4. Exercice D : Un type abstrait de données Arbre Binaire ... 37
7.2. BIBLIOGRAPHIE ... 38

Remarques de Daniel Feneuille (13/08/01) :

Le cours de testabilité était assuré depuis 1995 par Patrice Micouin professeur associé au département informatique. Notre collègue ne désirant plus reconduire son contrat je prends la relève pour immerger ce cours de 2 heures avec les autres cours Ada que j'assume déjà.

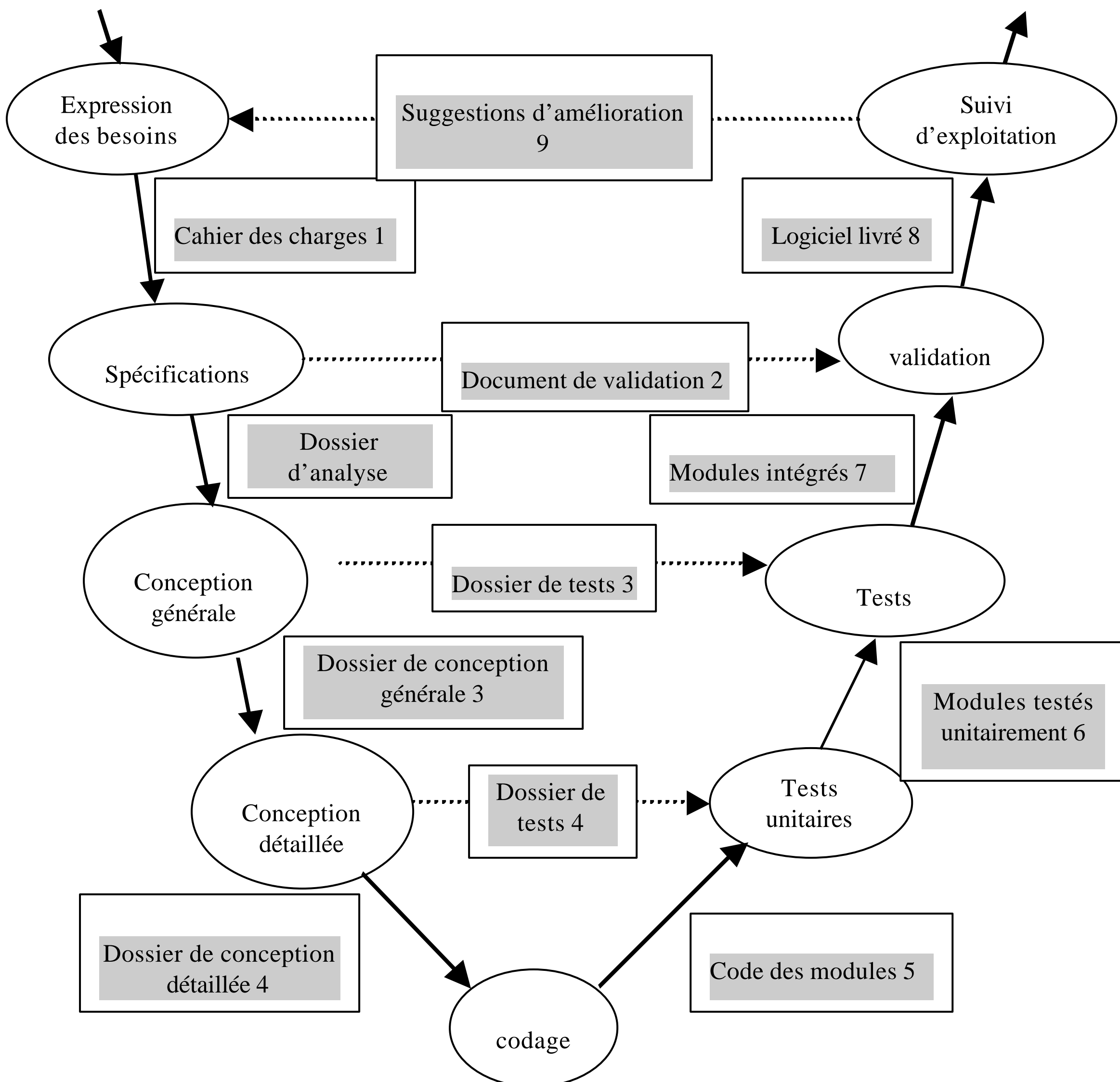
Le polycopié qui complète ce cours reste pratiquement inchangé et c'est celui-ci. J'ai seulement apporté quelques petits compléments pour être en phase avec les cours Ada notamment avec le cours 10 où l'on parle énormément de génie logiciel, de type abstrait de données, de conception orientée objet et de méthodologie.

Ce polycopié est, et restera, une référence importante pour le cours de génie logiciel. J'invite les étudiants à conserver ce document, il pourra leur être utile par la suite soit en deuxième année soit dans le cadre d'autres études d'informatique voire même dans la profession où il arrive que l'on mette en pratique « le test » sujet de cette étude.

Chaque année (comme d'ailleurs pour le cours 10 Ada) beaucoup d'étudiants ne goûtent pas assez la substantifique moelle des concepts enseignés. C'est dommage quoique normal tellement ces choses là sont nouvelles et bien éloignées du bidouillage que l'on affectionne au début des premiers apprentissages.

FICHE DE SUIVI DES EVOLUTIONS DU PRESENT DOCUMENT

Version	Date	Commentaires
1	Hiver 95	Création du document
2	Eté 95	Prises en compte des remarques de D. Feneuille, D. Mathieu et T. Avignon. - Introduction du guide du test boîte claire.
3	Hiver 96	- Introduction de la notion de test statistique - Introduction au test des logiciels temps réel - Programmes concurrents
4	Hiver 97	Refonte du plan - Introduction du test des objets - Introduction de la notion de scénario de test - Introduction du test des génériques
5	Automne 98	Evolution Ada 83 vers Ada 95 Mesure des temps d'exécution
6	Août 2001	Quelques ajouts de Daniel Feneuille : mise en évidence de certains concepts, recadrage par rapport au cours Ada notamment le n° 10 et aux TD-TP.



GENERALITES

1.1. QU'EST-CE QUE LE TEST ?

D'abord (**ce que n'est pas le test** !) :

"Le test **n'est pas un moyen** pour s'assurer qu'un programme fonctionne". Erreur fréquente !

Autres définitions **incorrectes** du test :

"Le test est le processus qui permet de démontrer que le programme est sans erreur."

"L'objectif du test est de montrer que le programme exécute correctement les fonctions prévues".

"Le test est le processus qui permet de s'assurer que le programme fait ce qu'il est supposé faire".

Au contraire, le test est un processus <u>agressif</u> dont le but est de <u>trouver</u> des erreurs dans un programme.

Par conséquent, un bon **test est un essai qui trouve des erreurs** et pas un essai qui dit que tout va bien.

1.2. TESTER N'EST PAS DEBUGGER

Le mot "test" est un anglicisme. *To test = essayer*. Tester veut donc dire <u>procéder à des essais</u>.

L'opération de débuggage (déverminage ou mise au point) est une opération qui vient juste après le codage et vise à faire en sorte que le code s'exécute à peu près correctement.

Le test est une opération qui vient **après** le débuggage et son but déclaré est inverse (trouver des erreurs).

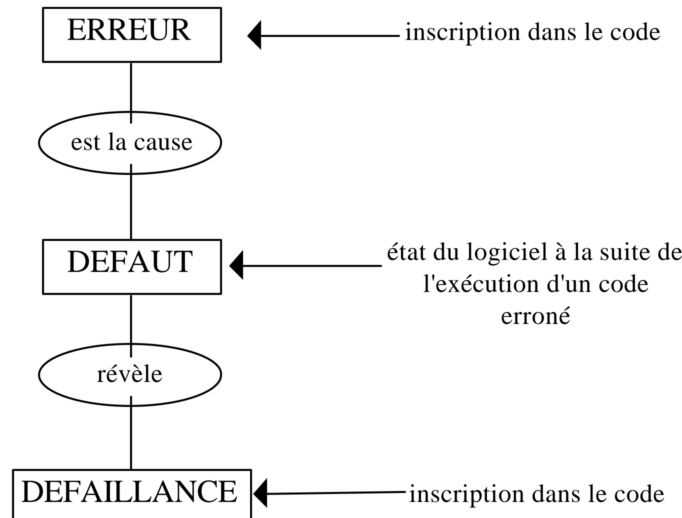
1.3. QUI TESTE ?

L'auteur d'un programme n'est pas la personne la mieux placée pour le tester. En effet, le réalisateur d'un programme aura une tendance naturelle à épargner son programme (narcissisme ? !).

Ainsi en aéronautique, le test des logiciels critiques[1] doit être réalisé par des équipes différentes des équipes de développement : c'est une des formes de la **vérification indépendante**.

1.4. QU'EST-CE QU'UNE ERREUR ?

Une **erreur** est une inscription dans le code qui va entraîner un **défaut** lors de l'exécution de ce code. Ce défaut pourra être à l'origine d'une **défaillance**[2].



Une défaillance est un **écart** constaté du comportement logiciel par rapport au comportement tel qu'il a été spécifié[3].

---

[1] Dans la norme américaine DO 178 version A, un logiciel aéronautique est dit critique si sa défaillance engage directement la survie des personnes transportées ou survolées, par exemple le Pilote Automatique Numérique ou les Commandes de Vol.

[2] Certains auteurs, comme Laprie, inversent les définitions d'erreur et de défaut.

[3] On notera que cette spécification peut être partiellement implicite, par exemple, le fait que le logiciel soit **robuste** fait partie de ses spécifications implicites.

2. LE TEST D'UN SOUS PROGRAMME

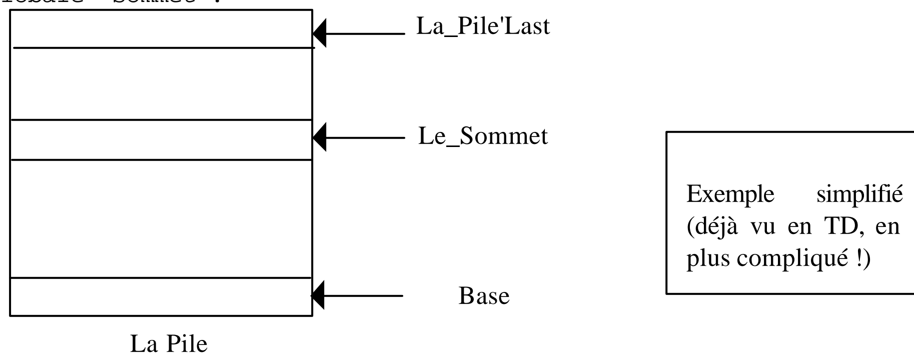
2.1. ENVIRONNEMENT DE TEST

Un sous programme est :

* une unité de petite taille (100 lignes au plus)
* admettant des paramètres en entrée et en sortie (données, résultats, données-résultats),
* agissant éventuellement sur des "variables globales" (rappel : danger !)
* compilable séparément (ou presque[4]).

Spécification textuelle :

Ecrire une procédure permettant d'empiler un entier au sommet d'une pile d'entiers de hauteur bornée (non définie).
Lorsque la pile est pleine, l'exception Débordement doit être levée. La pile est une variable globale de type tableau et la dernière case occupée est pointée par la variable globale "Sommet".



Spécification Ada (simplifiée)

```
package P_PILE is
   type Pile (Taille : Natural) is private;
   procedure EMPILER (L_ELEMENT : in INTEGER; Dans : in out Pile);
--   procedure DEPILER (De : in out Pile; L_ELEMENT : out INTEGER);
   Pile_Pleine : Exception;
--   Pile_Vide: Exception;
private
   type Implementation_Pile is array (Natural range <>) of Integer;
   type Pile (Taille : Natural) is record
      Une_Pile: Implementation_Pile(0..Taille);
   end record;
   Le_Sommet : Natural;
end P_PILE;
```

Pour tester ce sous programme, on va définir un environnement de test : on va se placer dans un répertoire spécial dédié au test et dans lequel on trouve :

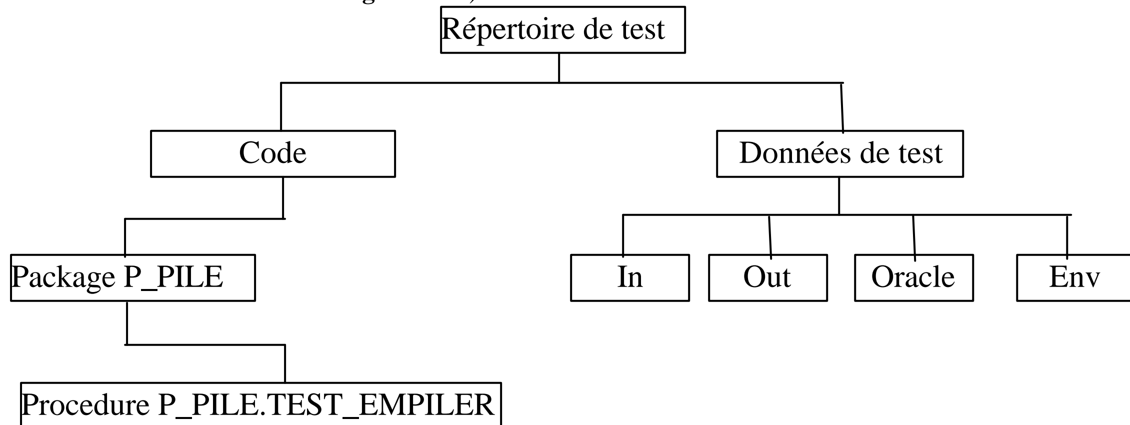
* le programme à tester,
* son "driver" de test (c'est un autre programme),
* un fichier contenant les variables d'environnement positionnées,
* un fichier contenant les valeurs en entrée,
* un fichier contenant les valeurs en sortie attendues,
* un fichier contenant les valeurs en sortie effectives.

En effet, il existe des règles d'or du test qui sont :

* On ne teste **jamais** un programme de **façon interactive**.
* Un jeu de test est un **capital** qui doit être **accumulé**

[4] Par adjonction d'un environnement limité.

* on doit pouvoir rejouer un test à tout instant et sans coût supplémentaire (cas notamment des **tests de non régression**[5])



- le procédure à tester (Empiler),

```
package body P_Pile is
procedure Empiler (L_Elément : in Integer; Dans : in out Pile) is
   begin
      if Le_Sommet < Dans.Une_Pile'Last then
         Le_Sommet := Le_Sommet + 1;
         Dans.Une_Pile(Le_Sommet) := L_Elément;
      else
         raise Pile_Pleine;
      end if;
   end Empiler;
end P_Pile;
```

- éventuellement des variables d'environnement

Fichier Env
3 -- Valeur du sommet

- un driver de test[6].

```
with Text_Io;
procedure Pile.Test_Empiler is
   File_In, File_Out, File_Env: Text_Io.File_Type;
   Taille : Natural;
   package Int_Io is new Text_IO.Integer_Io(Integer);
begin
   Text_IO.Open (File_Env, Text_IO.In_File,"Env.In");
   Int_Io.Get (File_Env, Taille);
   Text_IO.Close(File_Env);
   declare
      La_Pile : Pile(Taille);
      L_Elément : Integer;
   begin
      Text_Io.Create (File_Out, Text_IO.Out_File,"Fichier.Out");
      Text_IO.Open (File_In, Text_IO.In_File,"Fichier.In");
      while not (Text_Io.End_Of_File(File_In)) loop
         begin
               Int_Io.Get(File_In,L_Elément);
               Empiler (L_Elément,La_Pile);
```

[5] Les tests de non régression sont essentiels pour la maintenance d'un logiciel (voir définition plus loin). Un programme P a été développé avec son jeu de test T. Il est mis en opération. Si une erreur est découverte pendant cette période, on va réaliser une correction de l'erreur. Malheureusement, la correction d'une erreur en introduit souvent de nouvelles (c'est ce qu'on appelle la régression du logiciel). Pour s'en apercevoir, il faut rejouer T. On dit qu'on procède à des tests de non régression.

[6] La fabrication de driver est pratiquement automatisée grâce à l'utilisation d'outils de test.

```
          Text_Io.Put_Line(File_Out,"OK");
        exception
          when Pile_Pleine => Text_Io.Put_Line (File_Out,"NOK");
        end;
      end loop;
      Text_Io.Close(File_Out);
      Text_Io.Close(File_In);
    end;
end Pile.Test_Empiler;
```

- deux ou trois fichiers : un fichier en entrée contenant des données d'essai (jeu de test ou cas de test), un ou deux fichiers contenant l'un, les résultats obtenus lors d'une exécution et l'autre, les résultats théoriquement corrects (C'est ce qu'on appelle l'**Oracle**)

Fichier In
1
4
7
10

Oracle
OK
OK
OK
NOK

Fichier Out
OK
OK
OK
NOK

2.2. TYPES DE TEST

Un problème important du test concerne la sélection des jeux de test.

2.2.1. Test exhaustif.

Vouloir tester un programme exhaustivement, c'est-à-dire en essayant tous les cas possibles, est une illusion. Le test à 100% est impossible aussi bien en théorie qu'en pratique.

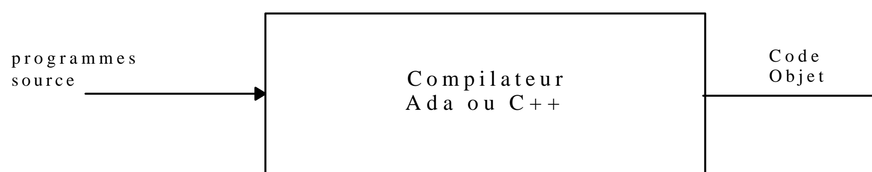
En pratique :

Ainsi sur une machine 16 bits, le test exhaustif de la simple instruction :

```
Somme := entier_1 + entier_2;
```

demanderait approximativement 2^{32} exécutions, c'est-à-dire que l'éternité n'y suffirait pas[7].

En théorie : Il est impossible de tester de façon exhaustive un programme (le nombre de cas de test possibles est infini).



Penser par exemple, au test d'un compilateur Ada ou C++, tout nouveau programme est un nouveau cas de test, et le nombre de programmes est infini (tous les programmes passés, présents et à venir).

2.2.2. Objectifs de la sélection des jeux de test.

Le test exhaustif étant impossible[8], on doit se **résigner** à un **ensemble limité de tests**, parce que cela :

* prend beaucoup de temps,
* coûte très cher.

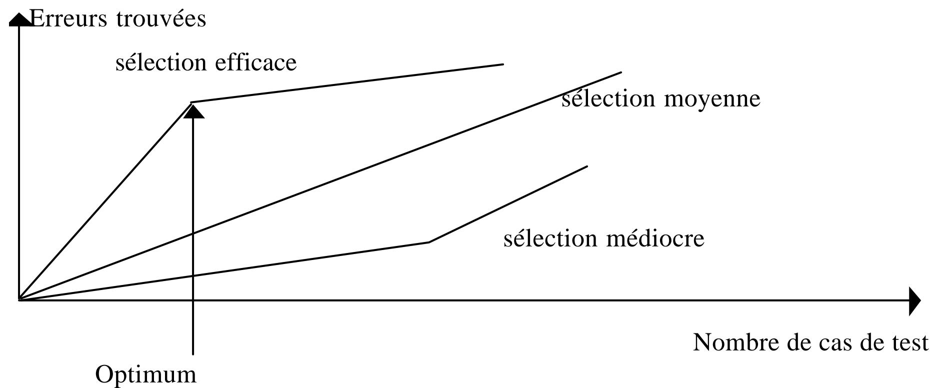
[7] Penser également au test exhaustif du programme récursif : "Les tours de Hanoi" (voir TD récursivité).

[8] Mais ce n'est pas pour autant qu'il ne faut pas tester !

Celui qui teste doit apporter une valeur ajoutée au programme qu'il teste. Cette **valeur ajoutée** ne peut être que la **découverte d'erreurs**.

Seul un ensemble fini de cas de test est envisageable. Cet ensemble peut être plus ou moins étendu et ce qui est recherché c'est un ensemble de cas de test (le plus réduit possible) qui mette en évidence le plus d'erreurs possibles.

Les jeux de tests doivent donc être choisis de telle sorte que pour un effort minimal, on obtienne une efficacité maximale. Un cas de test qui ne met pas à jour une erreur est un investissement en pure perte.



2.2.3. Test Statistique.

Principe : Les jeux de test sont tirés au hasard :

* utilisé dans le test du matériel,
* très controversé et peu utilisé en logiciel[9],
* pose le problème de l'oracle (prédiction du résultat juste).

Toutefois, on note depuis quelques années un regain d'intérêt pour ce type de test dont on trouvera une présentation ci-dessous en 2.4.5.

2.2.4. Typologie classique des tests.

On distingue deux grands types de méthode de sélection des jeux de test :

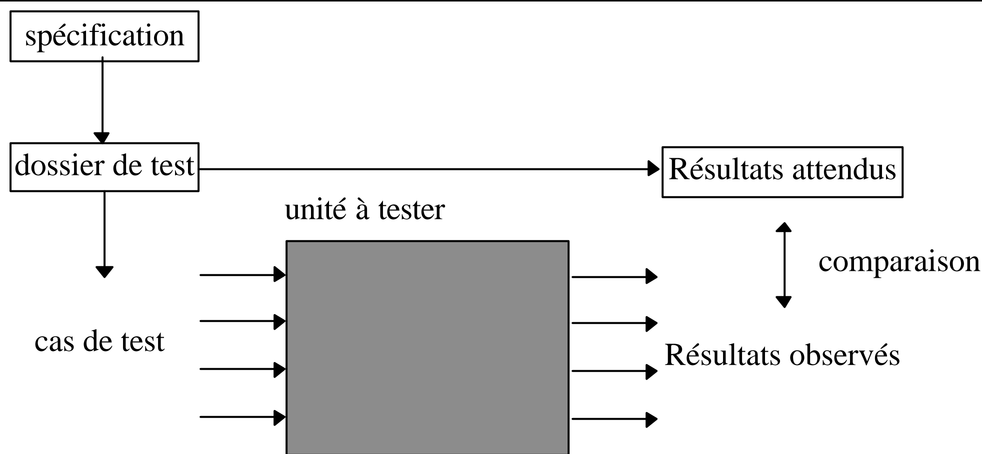
* les tests fonctionnels ou "boîte noire",
* les tests structurels ou "boîte claire".

2.2.4.1. Le Test "Boîte Noire"

Dans le test fonctionnel ou "boîte noire", on ne regarde pas COMMENT est construit le programme pour définir les jeux de test, on ne s'intéresse qu'à sa SPECIFICATION, i.e. ce qu'il est supposé faire (d'où le terme boîte noire, on ne regarde pas dedans).

Selon cette stratégie, le testeur voit le programme comme une boîte noire. Il ne s'intéresse pas à la structure interne du programme mais seulement aux spécifications du programme et son seul souci est de trouver des cas de test qui mettent en évidence un **écart** entre le comportement **réel** et le comportement **spécifié** du programme.

[9] Toutefois, certains auteurs ont montré que cette technique pouvait donner des résultats intéressants : J.W. Duran "An evaluation of random testing" IEEE TSE vol 10 n°4, july 1984.



Il n'est pas pensable de tester en essayant toutes les entrées possibles (infini), ce qui montre :

- qu'aucun processus de test[10] ne peut garantir qu'un programme est sans erreur,
- que la testabilité est d'abord une question économique et de prix que l'on est prêt à payer !

Avantage et inconvénient du test "boîte noire" :

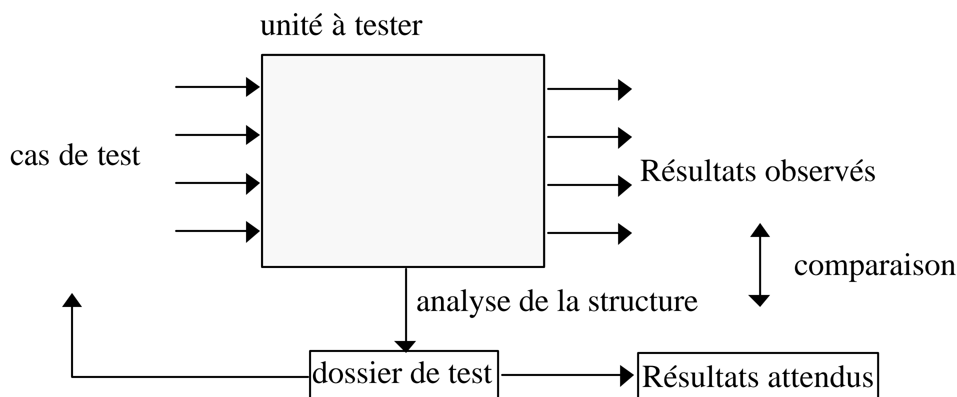
- inconvénient : le test "boîte noire" est moins facile à mettre en œuvre que le test "boîte claire"[11],
- avantage, si la structure du programme change, les jeux de test ne changent pas. Le test "boîte noire" est stable.

2.2.4.2. Le test "Boîte Claire"

Dans le test structurel ou "boîte claire" on bâtit les jeux de test en fonction de l'organisation du programme (dont on voit les sources).

Suivant cette stratégie, le testeur s'intéresse à la structure interne du programme et à sa logique (et parfois au détriment de sa spécification).

On va sélectionner des cas de test de façon à éprouver les structures et chemins de contrôle du programme (mais de la même manière que pour le test boîte noire, il est impossible de vouloir tester tous les chemins de contrôle).



[10] La preuve de programme est une voie alternative et complémentaire au test.

[11] ne serait ce que par ce qu'il suppose des spécifications explicites, ce qui n'est pas toujours fait.

Avantage et inconvénients du test "boîte claire" :

- avantage : Le test "boîte claire" est facile à mettre en œuvre,

- premier inconvénient, si la structure du programme change, les jeux de test changent également. Le test boîte claire est instable,

- autre inconvénient, on ne détecte pas l'absence de chemins nécessaires, ni d'erreurs dépendant des données.

2.3. - TECHNIQUES DU TEST STRUCTUREL (BOÎTE CLAIRE)

Le test "boîte claire" d'un programme met à jour sa structure interne. En particulier, l'observation du programme permet de savoir s'il s'agit d'un programme séquentiel ou d'un programme concurrent.

2.3.1. Le test structurel des programmes séquentiels

Pour le test "boîte claire", les techniques envisageables sont
- la couverture d'instructions
- la couverture de décisions
- la couverture de conditions
- la couverture décisions/conditions
- la couverture conditions/multiple (non décrit ici)

2.3.1.1. Les chemins de contrôle

Bloc séquentiel : Suite d'instructions telle que l'exécution de la première instruction entraîne celle de la dernière (sauf si levée d'exception).

Segment : Bloc séquentiel suivi du transfert de contrôle qui la termine.

Branche : Transfert de contrôle qui résulte d'une décision.

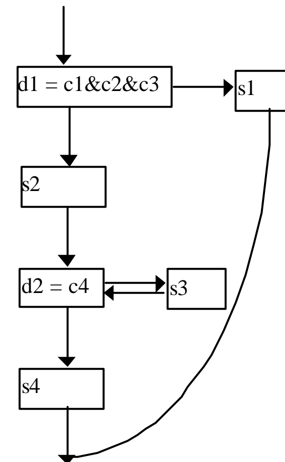
Chemin D-D : Chemin de décision à décision est une suite de nœuds du graphe de contrôle :
- commençant par le point d'entrée du programme ou par un nœud de décision,
- se terminant par le point de sortie du programme ou par un nœud de décision,
- ne comportant aucun nœud de décision entre les nœuds d'entrée et de sortie.

Chemin de contrôle : Séquence de nœuds du graphe de contrôle partant de l'entrée du programme et finissant à la sortie du programme.

Chemin d'exécution : Chemin de contrôle effectivement parcouru durant une exécution du programme. Un chemin de contrôle n'est pas forcément un chemin d'exécution.

Exemple :

```
function Racine_Carrée
          (A : in Float; Erreur : in Float)
          Return Float is
begin
if A>=0.0 and Erreur>0.0 and Erreur<1.0 then
   declare -- s2
      Précédent : Float := A/2.0;
      Suivant : Float := A;
   begin
   while Abs(Suivant- Précédent)>=Erreur loop
      -- s3
      Précédent := Suivant;
      -- Processus de Heron[12]
      Suivant := 1/2*( Précédent +A/ Précédent);
   end loop;
   return Suivant;
   end;
else
   raise Pre_Condition_Error; -- s1
end if;
end Racine_Carrée;
```



2.3.1.2. Couverture d'instructions

On définit des cas de test de telle sorte que toutes les séquences d'instructions du programme soient exécutées au moins une fois

Sur l'exemple précédent :

Cas de test	Résultat attendu	Séquences exécutées
A = -1.0, Erreur = 0.1	Exception levée	s1
A = 2.0, Erreur = 0.1	1.4166...	s2, s3, s4

Le test couvre toutes les instructions

2.3.1.3. Couverture de branches (décisions)

On définit des cas de test de telle sorte que toute les branches du programme soient parcourues au moins une fois. C'est une exigence plus forte que la précédente (couverture instructions).

Sur l'exemple précédent :

Cas de test	Résultat attendu	Décisions évaluées
A = -1.0, Erreur = 0.1	Levée d'exception[13]	~~d1~~[14]
A = 2.0, Erreur = 0.1	1.4166...	d1[15],d2,~~d2~~

Le test couvre toutes les branches

Remarque : l'exemple choisi ne permet pas de bien discerner la différence entre les couvertures d'instructions et de branches.

[12] Heron d'Alexandrie 1er siècle après JC (voir cours 2 numériques Ada).

[13] exception Pre_Condition_Error.

[14] la décision d1 est évaluée à faux.

[15] la décision d1 est évaluée à vrai.

2.3.1.4. Couverture de conditions

On définit des cas de test de telle sorte que toutes les conditions contenues dans chaque décis ion du programme prennent toutes les valeurs possibles au moins une fois.

Sur l'exemple précédent (Racine_Carrée) :

Cas de test	Résultat attendu	Conditions évaluées
A = -1.0, Erreur = -0.1	Levée d'exception	c1,c2,~~c3~~
A = 1.0, Erreur = 1.1	Levée d'exception	~~c1~~,~~c2~~,c3
A = 2.0, Erreur = 0.1	1.4166...	~~c1~~,~~c2~~,~~c3~~,c4,~~c4~~

Le test couvre toutes les conditions et toutes les branches. Mais cette exigence n'est pas plus forte que la précédente (couverture de décisions).

Il peut y avoir des cas où toutes les conditions sont évaluées une fois à V et à F sans que la décision soit évaluée dans les 2 configurations

Par exemple, si d1 = c1 and c2, les configurations (c1 , ~~c2~~) et (~~c1~~ , c2) permettent d'évaluer toutes les conditions sans évaluer d1 à V.

2.3.1.5. Couverture de condi tions/décisions

On définit des cas de test de telle sorte que toutes les conditions contenues dans chaque décision du programme prennent toutes les valeurs possibles au moins une fois et que toutes les décisions prennent toutes les valeurs possibles au moins une fois et que tous les points d'entrée[16] soient sollicités au moins une fois. Cette exigence n'est pas plus forte que la précédente (couverture décisions).

2.3.1.6. Itérations et récursivité : Quantités de contrôle

Les techniques de test structurel (boîte claire rappel !) présentées ci-dessus ne traitent pas deux sources importantes de défaillance :

- les boucles d'itération,
- les appels récursifs de sous-programmes.

En présence de structures telles que :

```
while condition loop
    suite_d_instructions;
end loop;
ou
loop
    suite_d_instructions;
    exit when condition;
end loop;
```

```
sous_programme SP (...) is
begin
    ...;
    SP(...);-- appel récursif de SP
    ...;
end;
```

quelles garanties a-t-on que **condition** sera évalué à faux (ou à vrai dans la seconde boucle) après un nombre fini d'itérations ou que **SP** ne sera pas appelé jusqu'à saturation de la mémoire (pile d'exécution)?

Pour minimiser les risques, on introduit souvent une **quantité de contrôle** Q(n) **entière** associée à la boucle ou au sous programme et qui vérifie les relations :

" n $\in$ I on a $0 < Q(n+1) < Q(n)$

C'est-à-dire **que la suite** $(Q(n))_{n \in I}$ est une suite strictement décroissante d'entiers positifs (ce qui garantit que l'ensemble I est fini).

[16]pour les tâches Ada par exemple

Ainsi dans le célèbre algorithme dit des "tours de Hanoï "

```
procedure Hanoi(N : in Natural ; X,Y,Z : in Socles) is
begin
    if N > 0 then
        Hanoi (N-1, X,Z,Y);
        Déplacer (X,Y);
        Hanoi (N-1, Z,Y,X)
    end if;
end loop;
```

Vu en TD récursivité !

la quantité de contrôle est bien sûr $(Q(n))_{n\in I}=(N-n)_{n\in\{1..N\}}$

2.3.2. Le test structurel des programmes parallèles[17]

Les programmes parallèles posent au responsable du test un problème significativement plus compliqué que les programmes séquentiels.

Pour illustrer cette complexité voici un exemple :

Considérons un programme contenant deux tâches Appelante et Appelée. Les problèmes qui peuvent apparaître sont alors les suivants :

- La tâche Appelante fait un appel à une entrée de la tâche Appelée alors que celle-ci est encore inactive,
- La tâche Appelante fait un appel à une entrée de la tâche Appelée alors que celle-ci est déjà terminée (normalement ou anormalement),
- L'une des deux tâches attend indéfiniment l'autre à un point de rendez-vous alors que l'autre ne s'y présente jamais,
- L'une des deux tâches ne se termine jamais,

Le test aura donc pour but de détecter une des situations précédentes

- Si les deux tâches ne sont pas élaborées dans le même cadre (master), le cadre de l'appelant est-il inclus dans le cadre de l'appelé ? (sinon possibilité de Tasking_Error),
- Existe-t-il des cas où l'appelée se termine avant l'appelant ? (possibilité de Tasking_Error),
- L'appelant peut-il attendre à une entrée tandis que l'appelée attend à une autre entrée? (inter-blocage),
- Existe-t-il des conditions d'arrêt des tâches ?

[17] voir le cours et les TD-TP correspondants qui sont programmés au début du deuxième trimestre.

2.3.3. La mesure des temps d'exécution

Un autre aspect important du test, lorsqu'il s'agit de faire un logiciel répondant à des exigences de temps de réponse est la mesure des temps d'exécution.

La réalisation des tests permet de vérifier si les performances attendues sont au rendez-vous. Pour cela il suffit d'instrumenter les drivers de test de la manière suivante :

```
with Text_Io;
with Ada.Calendar;
use Ada.Calendar;
procedure Pile.Test_Empiler is
   ..;
begin
   ..;
   declare
      ..
      Debut : Time;
      Fin : Time;
   begin
      ..;
      while not (Text_Io.End_Of_File(File_In)) loop
         begin
            ..;
            Debut := Clock; -- temps avant l'appel
            Empiler (L_Element,La_Pile);
            Fin := Clock; ; -- temps après l'appel
            Text_Io.Put(File_Out,"OK Duree    ");
         exception
            when Pile_Pleine => Text_Io.Put_Line(File_Out,"NOK Duree");
         end;
        Text_Io.Put_Line(File_Out, Duration'Image(Fin - Debut));
end loop;
      ..;
   end;
end Pile.Test_Empiler;
```

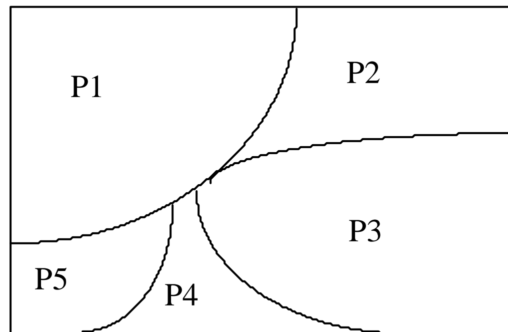
Cette mesure unitaire peut s'avérer infructueuse si l'horloge de la machine est insuffisamment fine et que Fin - Début = 0.0. Dans ce cas il faut calculer un temps moyen sur n (n= 1000 par exemple) appels du sous-programme.

2.4. TECHNIQUES DU TEST FONCTIONNEL (BOITE NOIRE)

Pour le test "boîte noire", les techniques envisageables sont
- le partitionnement
- l'analyse des valeurs-frontière
- les arbres causes/effets
- la devinette d'erreurs

2.4.1. La technique du partitionnement

La notion de relation et de classes d'équivalence a été vue en mathématiques ainsi que la notion de partition qui en résulte.



Domaine d'entrée

Ici E est l'ensemble des valeurs d'entrée possibles du programme (généralement infini) :

* Certaines de ces valeurs sont valides et le programme doit fournir les valeurs de sortie correctes,
* d'autres sont au contraire invalides mais le programme ne doit pas pour autant s'écrouler.

On va chercher dans la spécification du programme P les moyens de diviser le domaine d'entrée en classes d'équivalence de sorte que :
si a et b ∈ Classe(a), tester Prog pour la valeur a est équivalent à tester Prog pour b

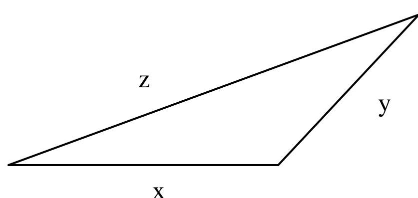
Exemple :

On a la spécification suivante :

Etant donné trois nombres réels a, b et c, écrire un fonction qui détermine si a, b et c sont les longueurs des trois côtés d'un triangle équilatéral, isocèle ou scalène.

La spécification Ada associée est :

```
type Nature is (Erreur, Equilatéral, Isocèle, Scalène);
subtype Longueur is float range 0.0 ..10.0;
function Nature_Triangle (a,b,c : in Longueur) return Nature;
```



Premier Prédicat C(x) : "x est la longueur d'un côté d'un triangle"

=> x > 0.

Second Prédicat T(x,y,z) : "x, y, z sont les longueurs des côtés d'un triangle" (double inégalité du triangle)

=> C(x) & | C(y) & C(z) & y-z|<x<y+z & |z-x|<y<z+x & |x-y|<z<x+y

Troisième Prédicat E(x,y,z) : "x, y, z sont les longueurs des côtés d'un triangle équilatéral"

=> C(x) & x=y=z.

Quatrième Prédicat I(x,y,z) : "x, y, z sont les longueurs des côtés d'un triangle isocèle"

=> T(x,y,z) & (x=y≠z ou y=z≠x ou z=x≠y).

	Valide	Invalides
C(x)	(3,4,5)	(2,-3,4)
T(x,y,z)	(3,4,5)	(3,4,8) (1,2,3)
E(x,y,z)	(3,3,3)	(3,3,2)
I(x,y,z)	(3,3,2)	(3,3,3) (3,3,4)

Heuristiques :

Heuristique sur la condition d'entrée	Valeurs valides	Valeurs invalides
x∈[a , b] x varie entre 0 et 10	a < x < b x=4	x < a et x > b x=-1 et x=12
x∈{ a , b , c , d } x est dans la liste {rouge, jaune, bleu}	x = c x = jaune	x = f x =vert
P(x) le premier caractère doit être une lettre	P(x) x = 'a'	non P(x) x = '_'
Si pour une raison quelconque on soupçonne que deux entrées "équivalentes" sont traitées différemment par le programme, alors dédoubler les classes d'équivalence		

2.4.2. Le test aux limites

La technique précédente n'est cependant pas suffisante. On s'aperçoit souvent que les programmes sont défaillants lorsqu'on les exécute aux bornes du domaine d'entrée. On va donc par conséquent essayer systématiquement le programme au voisinage des valeurs frontières.

Heuristiques :

Heuristique sur la condition d'entrée x ou de sortie y	Valeurs valides	Valeurs invalides
$x \in [a, b]$ x varie entre 0 et 10	x = a et x = b x = 0 et x = 10	x = a - ε et x = b + ε x = -0,001 et x = 10,001
$y = f(x) \in [a, b]$	chercher x tel que y = f(x)= a puis y = f(x) = b	chercher x tel que y = f(x) < a puis y = f(x) > b
Si une entrée ou une sortie est un ensemble ordonné	essayer sur le 1er et le dernier éléments	

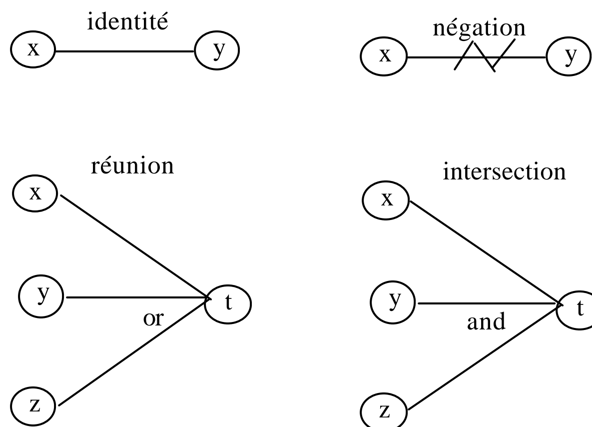
2.4.3. Les arbres causes/conséquences

C'est une technique qui tient son origine de l'analyse de sûreté.

Une **cause** est une condition d'entrée ou une configuration de conditions d'entrée.

Une **conséquence** est une condition de sortie ou une configuration de conditions de sortie.

Les causes et conséquences sont extraites de la spécification et organisées suivant un **arbre booléen causes/conséquences.**



2.4.4. La devinette

C'est une technique propre aux gens très expérimentés capables de "renifler" les situations fautives de manière intuitive.

Exemple :

Tester un algorithme de tri sur une liste

Essayer :

- une liste vide
- une liste déjà triée
- une liste avec un nombre pair d'éléments
- une liste avec un nombre impair d'éléments

2.4.5. Le test statistique

L'intérêt pour le test statistique tire son regain de l'observation suivante : Les techniques boîte noire classiques n'intègrent pas dans leur stratégie la façon dont le logiciel va être utilisé concrètement, si bien que l'effort mis pour tester peut être sans rapport avec les conditions réelles d'exécution du logiciel.

Par exemple : Un logiciel L fonctionne suivant trois modalités durant sa vie opérationnelle. Pour la modalité 1 les techniques BN ont permis de définir 300 cas de test et pour les deux autres modalités respectivement 500 et 200 cas de test.

Or le logiciel va fonctionner pendant sa durée de vie suivant ces trois modalités, (dont le test doit fournir **le même niveau de confiance**), dans les proportions suivantes modalité 1 : 5%, modalité 2 : 90% et modalité 3 : 5%. Résumons :

Modalité	Nombre de cas de test	Durée de vie
1	300	5%
2	500	90%
3	200	5%

L'effort de test de la modalité 2 est donc **proportionnellement très insuffisant** par rapport aux deux autres, si sa **criticité** est la même que celle des deux autres modalités.

L'idée de base du test statistique est donc la suivante :

L'effort de test consacré à un mode de fonctionnement doit être proportionnel à :

- La criticité de ce mode
- La probabilité d'être dans ce mode pendant la durée de vie du logiciel

2.4.5.1. Criticité d'un logiciel

La criticité d'un logiciel est définie par les conséquences d'une défaillance de celui-ci. Dans certaines normes (D0 178 A) on classe les logiciels en trois catégories :

- Critique : Les conséquences d'une défaillance sont catastrophiques pour la sécurité des biens et des personnes (par exemple, la défaillance du pilote automatique d'un avion),
- Essentiel : Une accumulation de défaillances peut avoir des conséquences catastrophiques pour la sécurité des biens et des personnes.
- Non-essentiel : Une défaillance n'a aucune incidence directe ou indirecte sur la sécurité des biens et des personnes.

L'effort de test devra être proportionnel à la criticité du logiciel.

2.4.5.2. Modèle d'usage d'un logiciel

Un modèle d'usage défini les modes de fonctionnement d'un logiciel et la durée de vie du logiciel dans ses différents modes.

Ainsi le logiciel de contrôle d'attitude d'un satellite peut fonctionner en plusieurs modes dont le mode Mise à poste, Normal et Fin de vie.

Mode	Durée de vie	Criticité	Effort de Test (nombre de cas de test)
Mise à poste	2%	Critique (risque de perte du satellite)	33%
Normal	85%	Essentiel (possibilité de contrôle à partir du sol)	56%
Autres modes	12%	Essentiel	8%
Fin de vie	1%	Non essentiel	epsilon

2.5. L'ARRET DES TESTS

Qu'il s'agisse de tests structurels (BL) ou fonctionnels (BN), le problème se pose de savoir quand l'effort de test peut-il s'arrêter, puisque, par nature, le test est **interminable**.

2.5.1. Arrêt des tests structurels

En ce qui concerne les tests structurels, la situation est assez simple puisqu'elle dépend du type de couverture que l'on s'est fixé.

- Si la couverture d'instructions a été choisie, le test est terminé lorsque 100% des instructions ont été couvertes par cette technique.
- Si la couverture de décisions a été choisie, le test est terminé lorsque 100% des décisions ont été couvertes.
- Si la couverture de conditions a été choisie, le test est terminé lorsque 100% des décisions ont été couvertes.
- etc ...

Remarque : Ce qui détermine le type de couverture choisi est la fiabilité requise du logiciel.

Par exemple, en se reportant à la norme DO 178A :

- pour un logiciel de criticité 3 (non-essentiel) la couverture **d'instructions** peut suffire,
- pour un logiciel de criticité 2b (essentiel) la couverture de **décisions** est nécessaire,
- pour un logiciel de criticité 2a (essentiel) la couverture de **conditions** est nécessaire,
- pour un logiciel de criticité 1 (critique) la couverture de **décisions**/**conditions** est nécessaire.

2.5.2. Arrêt des tests fonctionnels

La situation des tests fonctionnels est beaucoup moins claire car il n'existe pas de règle quantifiée (analogue au 100% de couverture du test structurel).

On peut alors procéder à la technique de l'**intrusion d'erreurs** encore appelée technique des **mutants**.

Soit P un programme et T l'ensemble des cas de tests fonctionnels qui ont servi à le tester. On envisage d'arrêter le test. Est-ce raisonnable ?

Pour le savoir, on fabrique n mutants du programme P. Chaque mutant P' du programme P se distingue de P par une modification atomique de son code.

Si par exemple, P contient l'expression
Condition_1 **and** Condition_2 on va faire la <u>mutation</u> Condition_1 **or** Condition_2 pour obtenir le programme P'.

Puis on va exécuter les n programmes mutants P' avec le jeu de test T du programme P. Chaque exécution avec un mutant doit normalement mettre en évidence une différence entre les résultats prévus et les résultats obtenus. Si ce n'est pas le cas, cela signifie que le jeu de test T n'est pas suffisamment discriminant et qu'il faut l'enrichir encore.

3. LE TEST DES MODULES

La section précédente a décrit les différents types de test d'une entité procédurale (fonction ou procédure). En fait, les nouvelles pratiques de programmation (programmation orientée objet, généricité) conduisent à considérer non plus le sous-programme comme élément de base à tester mais l'objet.

3.1.1. Le test d'un objet

Un objet est un ensemble de données masquées auxquelles sont associées des opérations de consultation et de modification.

Ces opérations peuvent être testées individuellement selon les techniques déjà présentées. Mais il se peut que ces opérations se coordonnent mal sur la structure de données qu'elles manipulent concurremment.

Par exemple, les services d'empilement et de dépilement d'une pile peuvent être en apparence individuellement corrects sans pour autant gérer de façon cohérente le sommet de pile, c'est-à-dire que la **composition** des opérations d'un objet doit être également testé. Cette composition doit en général respecter des **invariants**.

```
package PILE_ENTIERS is

   procedure Créer (Taille : in Positive);
   procedure Empiler (L_Element : in T_Entier);
   function Dépiler return T_Entier;
   function Est_Pleine return Boolean;
   function Est_Vide return Boolean;
   Débordement : exception;
   Initialisée : exception;
   Non_Initialiee : exception;
private
   ...;
end PILE_ENTIERS;
```

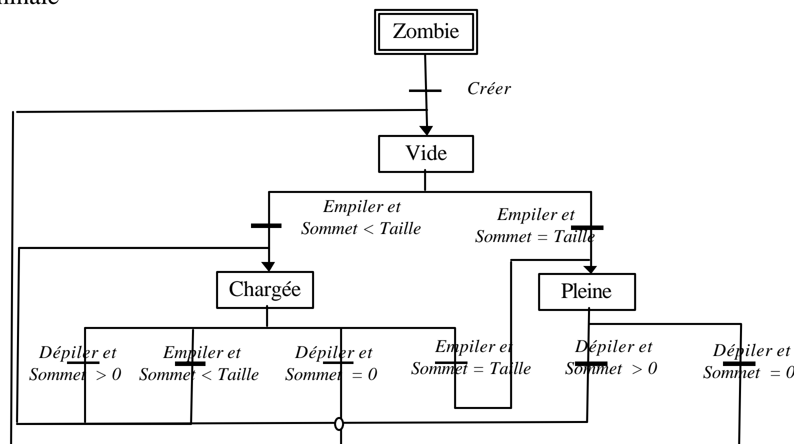
3.1.1.1. Comportement d'un objet

Le comportement de cet objet est caractérisé par :

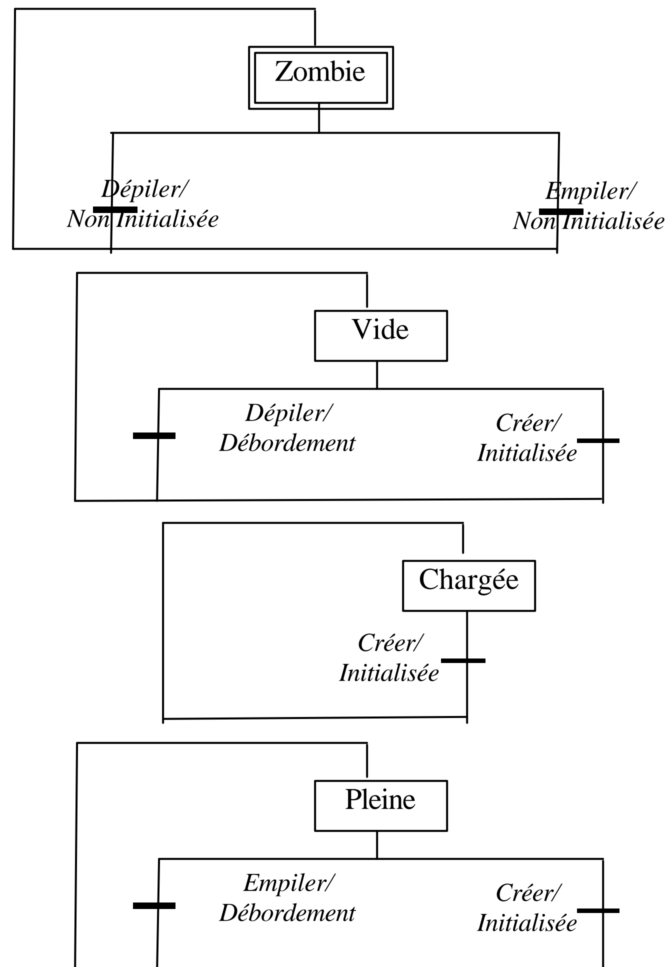
1. La nécessité d'initialiser sa taille en la créant avant la première utilisation,
2. L'impossibilité d'initialiser à nouveau la pile, une fois créée,
3. L'ordre inverse du dépilement par rapport à l'ordre d'empilement.

Ce comportement peut être modélisé à l'aide d'automates d'états finis ci-dessous. Le premier représente une utilisation nominale de l'objet tandis que les suivants expriment des utilisations non conformes de l'objet.

Utilisation nominale



Utilisations non conformes



3.1.1.2. Scénarios de test

Les scénarios de test sont des enchaînements de cas de test, appelant les divers services d'un objet destiné à éprouver le comportement de l'objet.

Ainsi pour l'exemple précédent, on est conduit à introduire des scénarios nominaux et des scénarios non conformes.

Scénarios nominaux

Scénario nominal avec Taille = 1

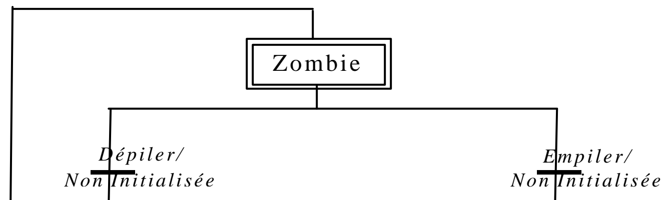
```
Créer (1); -- la pile est à un élément
Empiler (2); -- la pile est pleine
Retour (1) := Dépiler; -- la pile est à un élément
Comparer (2,Retour);
```

Scénario nominal avec Taille = 3

```
Créer (3); -- la pile est à 3 éléments
Empiler (2); -- la pile est chargée
Retour(1) := Dépiler; -- la pile est vide
Comparer (2,Retour);
Empiler (0); -- la pile est chargée
Empiler (2); -- la pile est chargée
Retour(1) := Dépiler; -- la pile est chargée
Retour(2) := Dépiler; -- la pile est vide
Comparer ((2,0),Retour);
Empiler (0); -- la pile est chargée
Empiler (2); -- la pile est chargée
Empiler (4); -- la pile est pleine
Retour(1) := Dépiler; -- la pile est chargée
Retour(2) := Dépiler; -- la pile est vide
Retour(3) := Dépiler; -- la pile est vide
Comparer ((4,2,0),Retour);
```

Il serait bon d'écrire également un scénario nominal avec Taille = 4

Scénarios non conformes

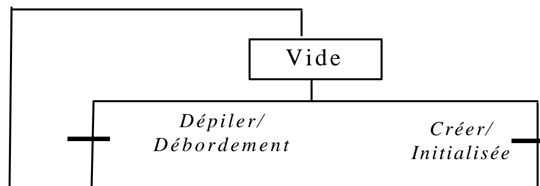


Scénario non conforme 1.1

```
Empiler (2); -- la pile n'est pas créée, exception non initialisée
```

Scénario non conforme 1.2

```
Retour(1) := Dépiler ; -- la pile n'est pas créée, exception non
     initialisée
```



Scénario non conforme 2.1

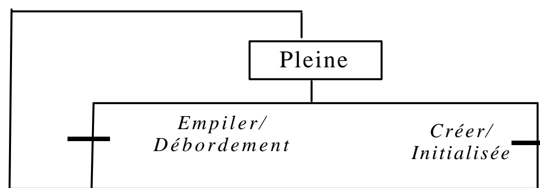
```
Créer (3); -- la pile est à 3 éléments
Créer (2); -- levée d'exception Initialisée
```

Scénario non conforme 2.2

```
Créer (3); -- la pile est à 3 éléments
Retour(1) := Dépiler ; -- la pile est vide, exception débordement
```

Scénario non conforme 3

```
Créer (3); -- la pile est à 3 éléments
Empiler (0); -- la pile est chargée
Empiler (2); -- la pile est chargée
Retour(1) := Dépiler; -- la pile est chargée
begin
   Créer (2); -- levée d'exception Initialisée
exception
   when Initialisée => null;
end;
Retour(2) := Dépiler; -- la pile est vide
Comparer ((2,0),Retour);
```



Scénario non conforme 4.1

```
Créer (2); -- la pile est à 2 éléments
Empiler (0); -- la pile est chargée
Empiler (2); -- la pile est pleine
begin
   Empiler (4); -- levée d'exception Débordement
exception
   when Débordement => null;
end;
Retour(1) := Dépiler; -- la pile est chargée
Retour(2) := Dépiler; -- la pile est chargée
Comparer ((2,0),Retour);
```

Scénario non conforme 4.2

```
Créer (2); -- la pile est à 2 éléments
Empiler (0); -- la pile est chargée
Empiler (2); -- la pile est pleine
begin
   Créer (3); -- levée d'exception Initialisée
exception
   when Initialisee => null;
end;
Retour(1) := Dépiler; -- la pile est chargée
Retour(2) := Dépiler; -- la pile est chargée
Comparer ((2,0),Retour);
```

3.1.2. Le test d'un générique

Il n'est pas possible de tester directement un générique, puisqu'on ne peut utiliser un générique que par l'intermédiaire de ses instances.

Dans un programme, on doit donc tester systématiquement toutes les instances d'un générique dès l'instant où les instances diffèrent par les paramètres d'instanciation . On pourra seulement se dispenser de tester deux fois deux instances ayant strictement les mêmes paramètres d'instanciation.

Un générique n'est jamais définitivement testé.

```
generic
   type ELEMENT is private;
package PILE is
   procedure Créer (Taille : in Positive);
   procedure Empiler (L_Element : in ELEMENT);
   function Dépiler return ELEMENT;
   function Est_Pleine return Boolean;
   function Est_Vide return Boolean;
   Débordement : exception;
   Initialisée : exception;
   Non_Initialisée : exception;
   Débordement : exception;
private
   ...;
end PILE;
```

Considérons maintenant deux instances de ce générique :

```
package PILE_ENTIERS is new PILE (Elément => Integer);
package PILE_REELS is new PILE (Elément => Float);
```

Alors les deux objets PILE_ENTIERS et PILE_REELS sont tous les deux à tester complètement

3.1.3. Le test d'une classe

L'équivalent Ada de la notion objet de classe est le type abstrait de donnée (en principe tagged !) défini dans un package comme par exemple :

```
package P_PILES_ENTIERS is
   type PILE_ENTIERS is private;
   procedure Créer (Une_Pile : in out PILE_ENTIERS; Taille : in Positive);
   procedure Empiler (Une_Pile : in out PILE_ENTIERS;L_Elément : in
        ELEMENT);
   function Dépiler(Une_Pile : in out PILE_ENTIERS) return ELEMENT;
   function Est_Pleine(Une_Pile : in PILE_ENTIERS) return Boolean;
   function Est_Vide(Une_Pile : in PILE_ENTIERS) return Boolean;
   Débordement : exception;
   Initialisée : exception;
   Non_Initialisée : exception;
   Débordement : exception;
end P_PILES_ENTIERS;
```

Le test de cette classe se fait par l'intermédiaire d'une déclaration d'objet dans un driver :

```
declare
   La_Pile : P_PILES_ENTIERS.PILE_ENTIERS;
begin

   scenarios_de_test;

end;
```

Si la classe (le type abstrait de données) est non générique, il suffit de tester seulement une instance, dans le cas contraire, il faudra tester toutes les instances n'ayant pas les mêmes paramètres d'instanciation.

4. LE TEST DES GRANDS PROGRAMMES

Le test des grands programmes n'est pas fondamentalement différent du test des modules. On applique toujours les mêmes techniques de test boîtes noire et claire.

La différence vient du fait qu'il n'est plus possible d'attaquer de front le test d'un grand programme et qu'il faut comme souvent "**diviser pour régner**".

On va donc découper le logiciel en "pièces humainement testables" (workable pieces) en faisant attention que l'intégration de ces pièces humainement testables soit également testable.

4.1. LES STRATEGIES DE TEST

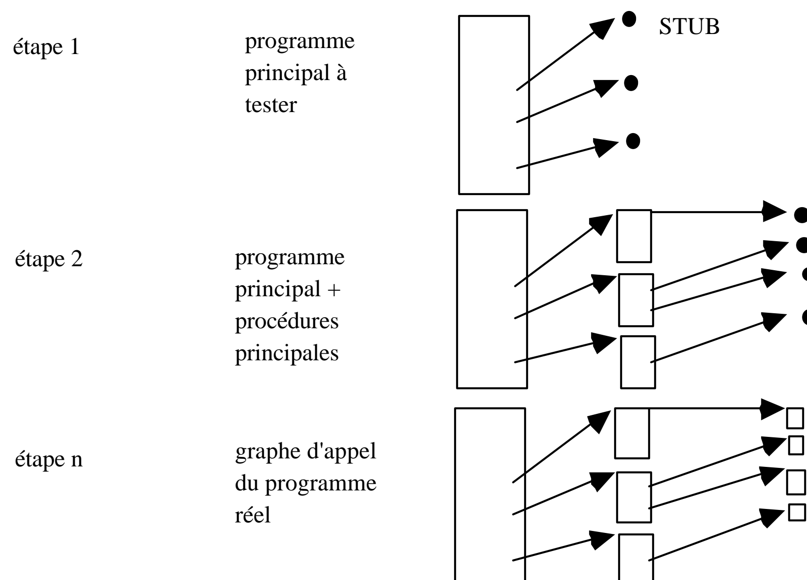
Pour procéder au test d'un grand programme deux démarches sont envisageables :

- la démarche descendante,
- la démarche ascendante,

4.1.1. Le test descendant (top-down)

La démarche descendante peut être décrite de la manière suivante :

- on commence par tester le programme principal en remplaçant les modules appelés par le programme principal par des émulateurs (**stubs**),
- on teste ensuite les modules appelés par le programme principal en remplaçant les modules appelés par des **stubs**.
- on réitère le processus jusqu'à atteindre les modules qui n'en appellent plus d'autres.

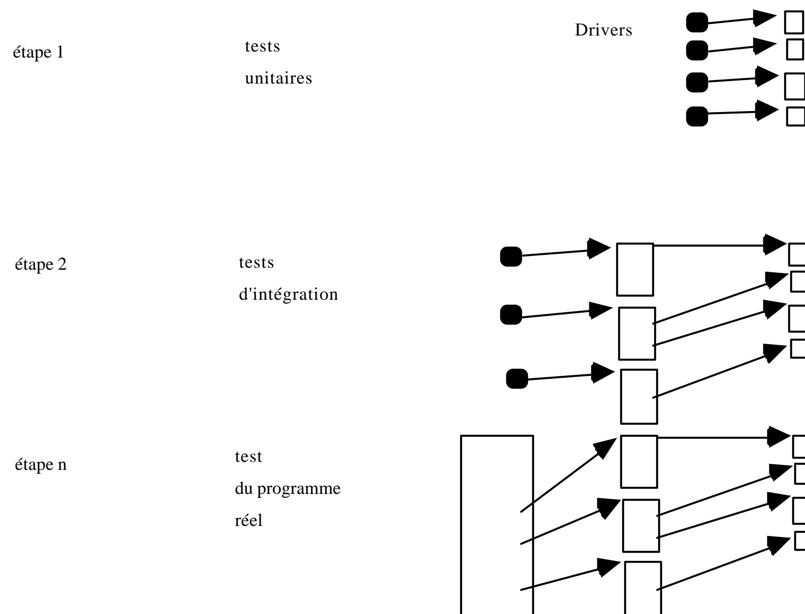


Cette stratégie de test est appréciée lorsqu'il n'y a pas eu de conception et que le logiciel a été directement codé.

4.1.2. Le test ascendant (bottom-up)

La démarche ascendante peut être décrite de la manière suivante :

- On commence par tester les modules[18] les plus bas dans la hiérarchie. Pour cela on doit introduire des pilotes (**Driver**) qui vont enchaîner les séquences de test.
- On teste ensuite les modules appelant des modules déjà testés en utilisant ou non des stubs.
- On réitère le processus jusqu'à atteindre le programme principal.



La démarche ascendante est dans la pratique la plus utilisée lorsqu'un véritable travail de conception a été réalisé.

Une variante de cette stratégie de test, appelée test incrémental, est généralement utilisée.

4.2. LE TEST INCREMENTAL

Le test incrémental vise à minimiser l'effort de test.

Son principe est simple :

Etant donné un module A, de deux choses l'une, soit il utilise des ressources d'autres modules, soit il est totalement indépendant.

Si A utilise les ressources des modules B1, B2, .., Bn indépendants les uns des autres,

-1- alors on teste individuellement d'abord B1, B2, .., Bn on parle alors de **tests unitaires**.

-2- Puis on teste l'ensemble (A, B1, B2, .., Bn) on parle alors de **tests d'intégration.** On procède ainsi jusqu'à obtenir l'intégration du logiciel complet L.

Si A est indépendant, il peut être testé unitairement.

4.2.1. Les tests unitaires

Les tests unitaires portent sur les modules testables individuellement.

Les tests **unitaires** sont plutôt des tests **structurels** (BC).

4.2.2. Les tests d'intégration

[18]Le module est ici selon les cas : sous programme ou objet.

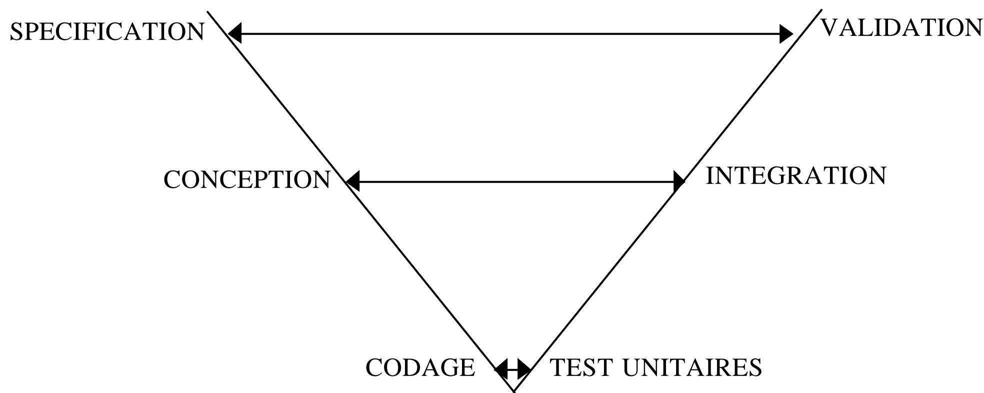
Les tests d'intégration portent sur des modules dépendant fonctionnellement d'autres modules.

Les tests d'intégration sont pour une part des tests structurels et d'autre part des tests fonctionnels.

4.2.3. Les tests de validation

Les tests de validation portent sur le logiciel complet. Ils ont lieu une fois que les tests d'intégration sont achevés.

Les tests de validation sont purement des tests fonctionnels. Ceci nous permet de rappeler une des figures les plus célèbres du génie logiciel : le cycle en V du développement du logiciel succinctement schématisé ci dessous (voir pour plus de détail voir le cours 10 Ada et la page 4 de ce document).



Tout logiciel doit être spécifié :
- sa spécification précise ce que doit faire le logiciel
- sa spécification permet la validation du logiciel

Tout logiciel (un peu compliqué) doit faire l'objet d'une conception :
- sa conception précise l'organisation du logiciel
- sa conception permet l'intégration du logiciel

© P. Micouin IUT d'Aix-en-Provence Département Informatique

5. LA TESTABILITE

5.1. QU'EST-CE QUE LA TESTABILITE

Comme on peut s'en douter, la testabilité est l'aptitude d'un code à être testé pour un **coût déterminé**.

Il ne faut pas perdre de vue que l'effort de test est **généralement plus élevé** que l'effort de codage et que selon la façon dont on s'y prend, l'effort de test peut devenir prohibitif.

Lorsqu'un code est mal écrit, il n'est pas rare qu'il soit intestable et il arrive que l'on soit obligé de le réécrire pour pouvoir le tester !!

5.2. TESTABILITE & COMPLEXITE

De manière générale, plus un logiciel est compliqué[19], plus il est compliqué à tester.

La complication est l'ennemi de la testabilité.

Il y a des problèmes naturellement compliqués, ou dont la solution est compliquée, mais si la conception est bien menée, la complexité du test reste maîtrisable (penser au test d'un compilateur seulement maîtrisable grâce à une bonne conception).

On trouve aussi des développeurs qui ont, soit par incompétence, soit par tournure d'esprit, tendance à construire des usines à gaz. Il faut les **neutraliser**.

Cette neutralisation s'obtient par :

- les **lectures croisées** de documents de spécification, de conception et de codage
- les **inspections** de documents de spécification, de conception et de codage.
- les **revues** de fin de spécification, de conception et de codage.

Avec le test, les revues, les inspections et lectures croisées sont des formes de la **vérification** de logiciel. La **preuve de programmes** (non traitée ici) est une autre forme de vérification.

5.3. TESTABILITE & SPECIFICATION

Un logiciel non spécifié ou insuffisamment spécifié est non testable.

On peut aussi dire de manière plus provocatrice d'**un logiciel non spécifié** qu'il **est exempt d'erreurs.**

C'est d'ailleurs ce qui se passe lorsqu'un fournisseur et un client se disputent à propos d'une livraison. Le fournisseur se réfugie derrière l'argument "*Ce n'était spécifié !*".

En effet, il n'est pas possible de définir les jeux de test fonctionnels d'un logiciel insuffisamment spécifié puisque ceux-ci s'en déduisent directement.

L'idéal est de disposer de **spécifications formelles** comme par exemple des **spécifications algébriques**. Il est possible de déduire automatiquement de ce type de spécifications des jeux de test.

Malheureusement, les techniques de spécification formelle n'étaient pas aptes à traiter des problèmes très compliqués. Très rapidement la complexité de la spécification peut devenir supérieure à la solution ("*un marteau pour écraser une mouche*"). Toutefois, certains programmes industriels à haut niveau de fiabilité sont aujourd'hui développés formellement. Le cas le plus illustre est celui des calculateurs embarqués à

[19] On utilise souvent comme synonymes **complexité** et **complication** (et c'est l'usage commun). Toutefois, il faut savoir qu'avec l'émergence de sciences comme la **systémique,** ces termes de complexité et de complication ont pris des chemins différents. Le fait qu'un organisme vivant puisse se régénérer relève de sa nature de système complexe (auto-organisation, auto-création) tandis que nous pauvres informaticiens, il faut bien nous rendre à l'évidence nous ne traitons en général que des problèmes compliqués (que l'on peut réduire à des algorithmes) sauf peut-être aux confins de l'intelligence artificielle. Le lecteur curieux pourra aller fouiller du côté des écrits de Edgard Morin, Prigogine, H. Atlan, Varela et bien d'autres encore, il suffit d'en prendre un et de tirer le fil.

bord des rames automatisées de la ligne 14 du métro parisien (la ligne METEOR), développé en langage formel B.

Il faut donc, dans la plupart des cas réels, se résoudre à des techniques de spécification moins formelles mais suffisamment précises tout de même pour pouvoir définir des **jeux de test de validation**.

A la fin de la phase de spécification, c'est-à-dire avant d'avoir réalisé le logiciel, on doit normalement être capable de produire le **plan de validation** du logiciel, c'est-à-dire la description de la façon dont on va **valider** (et **accepter**) le logiciel (voir cours 10 rappel !).

5.4. TESTABILITE & CONCEPTION

La conception décrit l'organisation (architecture) du logiciel pour répondre au problème posé par la spécification.

Cette architecture comprend une description précise :

- des **composants** du logiciel.
- des **interactions** entre les composants du logiciel.

La stratégie de test d'intégration va épouser l'architecture du logiciel. Par conséquent, un logiciel sans architecture est non intégrable.

Si le logiciel ressemble plus à un magma qu'à un cristal de roche, pour prendre une analogie avec la cristallographie ou si les interactions relèvent davantage du plat de spaghetti que d'un réseau structuré, il y a peu de chance que l'on puisse définir une stratégie de test d'intégration efficace et économique.

Concevoir un logiciel testable c'est :

- isoler des composants les plus indépendants les uns des autres que possible. Par exemple, **prohiber les variables globales**.
- assigner à chaque composant un **rôle** clair et **fortement cohérent**.
- définir pour chaque composant des **interfaces** fournies et requises aussi réduites que possible.
- définir entre les composants un flot de contrôle et de données minimal (**couplage faible**). Par exemple, on évite les **cycles** dans les graphes d'appel, la **récursivité** est souvent **interdite**.
- de façon générale, faire testable c'est faire **simple**, (les usines à gaz sont des catastrophes au niveau du test).

A la fin de la phase de conception, c'est-à-dire avant d'avoir réalisé le codage, on doit normalement être capable de produire le **plan d'intégration** du logiciel, c'est-à-dire la description de la façon dont on va, **étape par étape,** rassembler les différents éléments du logiciel en sous-systèmes testés.

5.5. TESTABILITE & CODAGE

5.5.1. Les critères de complexité

Au niveau du codage, il est possible de définir des critères quantitatifs de complexité. On doit coder des "pièces de programme ou modules", qu'il s'agisse d'une procédure ou d'une classe d'objets, répondant à des critères mesurables.

5.5.1.1. La taille

Le premier critère qui vient à l'esprit est la taille du module.

Il est par exemple absurde d'écrire une procédure faisant 1000 lignes.

On va donc se fixer un critère de taille, les **opérations** (procédures et fonctions) ne devront pas excéder **50** lignes hors commentaires, les **classes** ne devront pas excéder **500** lignes hors commentaires.

Ces chiffres sont purement indicatifs, il n'y a pas de règle absolue et chaque projet doit se fixer des règles du jeu et les respecter.

5.5.1.2. Le nombre cyclomatique

En fait la taille du module est un critère insuffisant parce qu'il ne mesure pas la complication du module en question. On peut avoir 50 lignes de code très simples (une séquence) ou au contraire 50 lignes très compliquées avec de nombreuses itérations, de la récursion et des alternatives, le tout fortement imbriqué.

Pour une opération Op donnée, si N est le nombre de nœuds, A et le nombre d'arcs alors le nombre cyclomatique est donné par la formule :

$C = A - N + 2$.

par exemple :

```
procedure EMPILER (L_ELEMENT: in INTEGER) is
begin
 if Le_SOMMET < La_PILE'LAST then
  Le_SOMMET := Le_SOMMET + 1;
  La_PILE(Le_SOMMET) := L_ELEMENT;
 else
  raise DEBORDEMENT;
 end if;
end EMPILER;
```

a pour nombre cyclomatique $C = 2$

5.5.2. Les conditions de passage en phase de test

Pour passer en phase de validation il faut que :

- la concordance de la conception vis-à-vis de la spécification ait été vérifiée (ceci est généralement obtenu lors de la **revue de conception**),
- les tests d'intégration soient achevés,
- les scénarios de validation aient été spécifiés sur la base du plan de validation.

Pour passer en phase d'intégration il faut que :

- la cohérence du codage à la conception ait été vérifiée (ceci est généralement obtenu lors de la **revue de codage**).
- les tests unitaires (concernant les parties à intégrer) soient achevés.
- les scénarios d'intégration aient été spécifiés sur la base du plan d'intégration.

Pour passer en phase de tests unitaires il faut que :

- le code source ait été accepté du point de vue de la testabilité (mesures de complexité concluantes).
- les scénarios de test unitaires aient été spécifiés sur la base du plan de tests unitaires.

6. OUTILS DE TEST

6.1. OUTILS D'ANALYSE STATIQUE

Les outils d'analyse statique sont plus des outils de qualimètrie que des outils de test.

Ces outils permettent de mesurer le niveau de complexité d'un code et en fonction de critères choisis de déterminer s'il est acceptable ou non (du point de vue de la testabilité).

Le plus connu de ces outils, en France a été Logiscope statique de la société Verilog, mais il en existe bien d'autres.

6.2. OUTILS D'ANALYSE DYNAMIQUE ET DE COUVERTURE

Les outils d'analyse dynamique et de couverture fournissent un environnement pour les tests "boîte claire".

Généralement le code est instrumenté de compteurs qui permettent de tracer les exécutions.

Ainsi pour un jeu de test donné on sait exactement, les instructions, branches ou conditions évaluées en fonction du niveau de test choisi.

6.3. SONDES ET EMULATEURS

Les sondes et émulateurs permettent de remplacer le calculateur et d'exécuter le logiciel comme s'il se trouvait sur le calculateur cible alors qu'il s'exécute sur le calculateur hôte (en général beaucoup plus convivial).

6.4. BANCS DE TEST

Les bancs de test sont des dispositifs qui permettent d'émuler l'environnement d'un système (calculateur+logiciel). C'est-à-dire que l'environnement émulé va envoyer au calculateur des trains de stimuli (événements+données) et va en retour mémoriser les réactions du calculateur à ces stimuli.

7. GUIDE DU TEST BOITE CLAIRE

Le but de cette section est de présenter de façon <u>non exhaustive</u> ce qui est à tester lorsqu'on examine la structure d'un programme Ada.

<table>
<tr><th>Instructions</th><th>Structure</th><th>Guide</th></tr>
<tr><td>Affectation</td><td>A := b;</td><td>Y-a-t-il des valeurs de b qui violent une contrainte de A ?</td></tr>
<tr><td>Expressions</td><td></td><td></td></tr>
<tr><td>Alternatives</td><td>if condition then
instructions;
[else instructions;]
end if;</td><td>Déterminer un cas où la condition est vraie et un cas où elle est fausse.</td></tr>
<tr><td></td><td>if c_1 then
instructions;
{elsif c_i then
instructions;}
[else instructions;]
end if;</td><td>Déterminer un cas où
c_1 est V,
c_1 est F, c_2 V
c_1, c_2 sont F, c_3 V
...
c_1, c_2, .., c_n-1 sont F, c_n V
c_1, c_2, .., c_n sont F.</td></tr>
<tr><td></td><td>case expression is
when choix_1|..|choix_i =>
instructions;
{when choix_j|..|choix_k =>
instructions;}
end case;</td><td><u>niveau 1</u> :
Déterminer un cas où
choix_1 .. ou .. choix_i est V,
...
choix_j .. ou .. choix_k est V.
<u>niveau 2</u> :
Déterminer un cas où
choix_1 est V puis,
..
choix_i est V puis,
...
choix_j est V puis,
..
choix_k est V.</td></tr>
<tr><td>Itérations</td><td>while condition loop
instructions;
end loop;</td><td>Examiner les cas où condition est V ou F.
En particulier est-on sûr que pour chaque itération condition deviendra F au bout d'un nombre fini d'itérations ? Exhiber la quantité de contrôle.</td></tr>
<tr><td></td><td>for id in [..] loop
instructions;
end loop;</td><td>Examiner tous les cas si possible ?</td></tr>
<tr><td></td><td>loop
instructions;
...
exit when condition;
instructions;
end loop;</td><td>Examiner les cas où condition est V ou F. En particulier est-on sûr que pour chaque itération condition deviendra V au bout d'un nombre fini d'itérations.</td></tr>
<tr><td>Débranchement</td><td>exit nom_boucle
when decision;</td><td>un cas pour decision = vrai
un cas pour decision = faux
(ça peut-être le même cas mais sur des itérations successives).</td></tr>
<tr><td></td><td>return expression;</td><td>néant</td></tr>
<tr><td></td><td>goto nom_etiquette;</td><td>néant</td></tr>
</table>

7.1. Exercices

7.1.1. Exercice A : Analyse syntaxique d'un identificateur

1- Ecrire une fonction booléenne Ada qui évalue si une chaîne de caractères est un identificateur ou non

function Est_Identificateur (Le_Mot : in String) return boolean;

2 - Tester fonctionnellement cette fonction

3 - Tester structurellement cette fonction

Voir le BNF complet dans le cours Ada n° 1

On rappelle qu'un identificateur obéit à la syntaxe suivante :
R 1: Identifier ::= letter { letter | digit | underline },
R 2 : un identificateur ne peut contenir deux underline successifs,

7.1.2. Exercice B : Un algorithme d'insertion dans un arbre binaire

Environnement du problème :

On considère une structure de données de type arbre binaire trié éléments d'un ensemble ordonné. Conceptuellement, c'est une structure composée de nœuds organisée de la manière suivante:

Tout nœud peut "pointer" sur au plus deux autres nœuds dits **gauche** et **droit**.

Si N1 "pointe" à gauche sur N2 et à droite sur N3 alors on a la relation d'ordre strict **N2.Value < N3.Value.**

Il existe un nœud particulier qui n'est pointé par aucun autre nœud, c'est la **racine**.

Position du problème :

On veut écrire un algorithme d'insertion qui maintienne ordonnée cette structure de donnée Arbre Binaire , c'est-à-dire :

Si on veut insérer El dans l'arbre A,

- **si El < A.Value alors on insère El dans le sous-arbre A.Gauche**
- **si El = A.Value on ne fait rien**
- **si El > A.Value alors on insère El dans le sous-arbre A.Droit**

Pour des raisons d'efficacité (l'allocation dynamique de mémoire est généralement interdite sur les logiciels temps réel critiques) on veut implanter un arbre binaire sous forme de tableau alloué statiquement de même qu'on s'interdit la récursion.

La spécification de cet algorithme est :

procedure Insérer (A_Value : In Natural; The_Tree : in out Tree);

S1 Si la valeur à insérer est déjà dans l'arbre, arbre n'est pas modifié

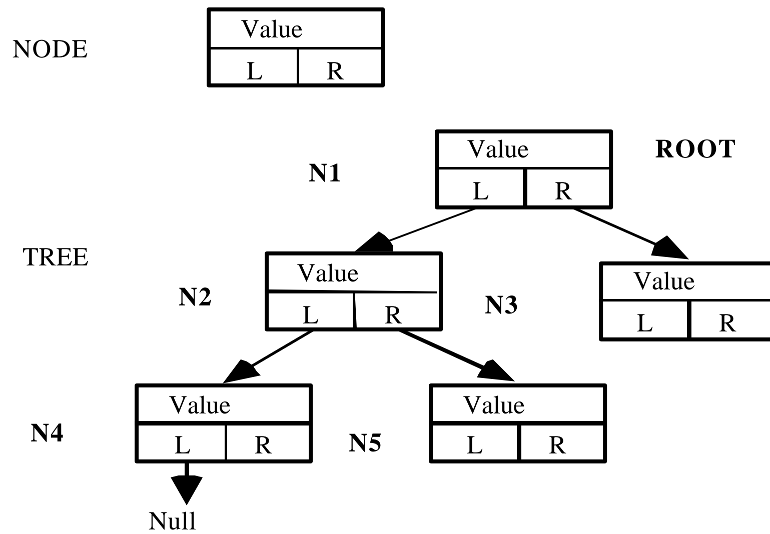
S2 A chaque insertion possible, la structure d'ordre de l'arbre doit être respecté et les règles précédentes respectées.

S3 Si la table qui implante l'arbre est pleine l'exception Tree_Overflow doit être levé

Question 1 : Ecrire cet algorithme Inserer

Question 2 : Tester structurellement Inserer avec 100% de couverture des décisions.

Question 3 : Tester fonctionnellement Inserer (technique de partition).



7.1.3. Exercice C : Le Calcul d'une intégrale par la méthode des trapèzes

On est fréquemment amené à interpoler, calculer des intégrales simples et résoudre numériquement des équations différentielles dans les applications techniques notamment pour les applications de simulation.

On veut intégrer numériquement une fonction f ne possédant pas de primitive connue entre les valeurs a et b de x.

La méthode consiste à partitionner [a , b] en n segments et d'approcher

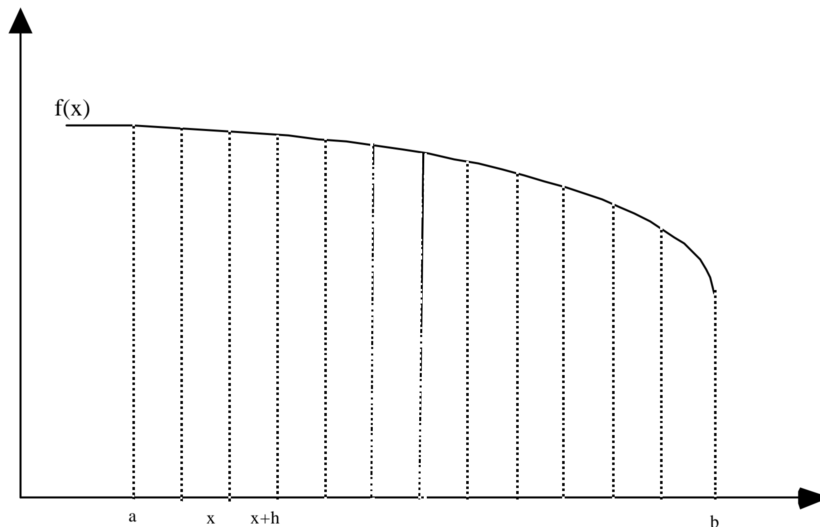
$$I = \int f(x)dx \text{ par } I' = h/2\ (f(a) + f(b) + 2\sum f(xi)\ )$$

avec h = (b-a)/n, xi = a+ i*(b-a)/n, la somme intégrale ∫ va de a à b et la somme discrète ∑ va de i= 1 à n-1

Question 1 : Ecrire cet algorithme Intégrer

Question 2 : Tester structurellement Intégrer (100% conditions)

Question 3 : Que pensez vous du test fonctionnel de cette fonction (partition+ limites).



Voir TD-TP numérique où cet exercice est entièrement résolu.

7.1.4. Exercice D : Un type abstrait de données Arbre Binaire.

L'exercice D est une extension de l'exercice B.

On veut définir un type abstrait de donnée (ce qui correspond à la notion de classe simplifiée).

La spécification Ada de cet ADT est

```
package Dictionary is
   type Word is new String;
   type Tree is private;
   procedure Insert (A_Value : in Word; A_Tree : in out Tree);
   procedure Edit (A_Tree : in Tree);
   function Search (A_Value : in Word; A_Tree : in Tree) return Boolean;

   private
      type Tree is ...;
end Dictionary;
```

La structure de données a les mêmes spécifications qu'en B (et les mêmes contraintes d'implantation).

La procédure d'insertion a les mêmes spécifications qu'en B

La procédure d'édition permet d'éditer l'ensemble des mots contenus dans le dictionnaire dans l'ordre lexicographique.

La fonction de recherche a des spécifications évidentes

Question 1 : Coder le corps de ce paquetage après avoir décidé d'une structure informatique efficace pour représenter les arbres binaires (ça c'est de la conception détaillée)

Question 2 : Définir le nombre cyclomatique de chaque méthode de la Classe Dictionary

Question 3 : Tester structurellement la Classe Dictionary

Question 4 : Tester fonctionnellement la Classe Dictionary

7.2. Bibliographie

[1] **The Art of Software Testing**, Auteur : Glenford J. Myers, Editeur : J. Wiley & Sons , 1979. C'est l'ouvrage de référence sur le test.

[2] **Le Génie Logiciel et ses applications**, Auteur : I. Sommerville, Editeur : InterEditions , 1988. Le chapitre 7 est consacré aux tests.

[3] **La maîtrise du développement de logiciel**, Auteur : B. Likov & J. Guttag, Editeur : Editions d'Organisation , 1990. Le chapitre 9 est consacré aux tests.

[4] **Les techniques de test structurel** , Auteurs : Ngo Pham Van & B. Amar, Revue : Génie Logiciel & Systèmes experts , n° 18, mars 1990. Numéro consacré aux tests.